UNIVERSITY of CALIFORNIA
Santa Barbara

Global-Scale Data Management with Strong Consistency Guarantees

A Dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Faisal Nawab

Committee in Charge:

Professor Divyakant Agrawal, Co-Chair
Professor Amr El Abbadi, Co-Chair
Professor Tevfik Bultan
Professor Xifeng Yan

January 2018

The Dissertation of Faisal Nawab is approved.

Professor Tevfik Bultan

Professor Xifeng Yan

Professor Amr El Abbadi, Committee Co-Chair

Professor Divyakant Agrawal, Committee Co-Chair

July 2017

Global-Scale Data Management with Strong Consistency Guarantees

by

Faisal Nawab

To my family.

Acknowledgements

My utmost gratitude goes to my advisors, Divy Agrawal and Amr El Abbadi. Thanks to them, I will leave graduate school set to pursue what I did not dare to dream of a few short years ago. From where I am to where I will be, I owe my thanks to them. Their unwavering passion for science has been a constant source of inspiration as I continue to grow as a researcher. They encouraged me to dive deeper into my research with unbelievable ability. I have always been in awe of their patience as we work to peel back the layers and reveal the core principles of a problem in our quest for the solution. Divy and Amr have taught me and my colleagues to be better researchers and educators. They inspire and lead by example. Their dedication to our growth is exemplary and I am fortunate to have lived these lessons with them.

I am thankful to Dr. Terence Kelly for his constant support, mentorship, and the opportunity to spend two years as a research associate at the renowned Hewlett-Packard Labs. Terence encouraged me to push myself and never fear taking my ideas to the next level. I will continue to follow Terence's example of nurturing a sharp, critical mind that is always ready to take on a challenge.

I am also thankful to Dr. Dave Lomet for his mentorship during my time as a research intern in the database group at Microsoft Research (MSR). Working at MSR and witnessing the professional, high-quality work of world-class researchers has been a transformative experience. Dave's mentorship has largely defined who I am as an academic. I aspire to imitate his relentless discipline in understanding research problems and developing technical solutions from end to end.

Dr. Basem Shihada has been a source of never-ending professional and personal support. His mentorship, while I was a student in KAUST, has been pivotal to my research career. From him, I learned the value of dedication and persistence. I am proud to be the first student he mentored, and I will always be grateful for his continuous

support.

I would like to acknowledge Professors Tevfik Bultan and Xifeng Yan for serving on my dissertation committee. Their valuable feedback throughout my graduate studies taught me maturity and inspired my work. I am also thankful for the opportunity to work with Professors Michael Scott from the University of Rochester and Sanjay Chawla from the University of Sydney. Their input and contributions to my work have been essential to giving it (and myself) an interdisciplinary breadth.

I have been fortunate to work with many other great researchers during my time in graduate school. Collaborating with Joseph Izraelevitz while we were at Hewlett-Packard Labs has been a great pleasure. His drive and ethics have been an inspiration to me. At UCSB, I enjoyed the discussions and work with Vaibhav Arora, Victor Zakhary, Alexander Pucher, Aaron Elmore, Cetin Sahin, Hatem Mahmoud, Theodore Georgiou, Xiaofei Du, and Sujaya Maiyya. Drs. Charles Morrey, Dhruva Chakrabarti, Harumi Kuno, Hideaki Kimura, Justin Levandoski, and Sudipta Sengupta of MSR and Hewlett-Packard Labs have provided me with great feedback and insight. I also thank Muath Alkhalaf for his friendship and our many candid discussions about research, formal methods, and distributed systems.

There is no greater source of support than that of my father and mother. They are my ultimate inspiration and source of strength. My sincerest thanks to Fahd, Nada, and Shada, my first teachers and companions at home.

I am most grateful for the daily love and encouragement from my wife, Reem. Words cannot describe how lucky I am to have her in my life. She has always been unconditionally supportive, pushing me to greater successes. Her love for science and dedication to becoming a better researcher are contagious. Her touch, influence, and elegant thoughts are in many, if not all, of the works in this dissertation. Thank you.

Abstract

Global-Scale Data Management with Strong Consistency Guarantees

by

Faisal Nawab

Global-scale data management (GSDM) empowers systems by providing higher levels of fault-tolerance, read availability, and efficiency in utilizing cloud resources. This has led to the emergence of global-scale data management and event processing. However, the Wide-Area Network (WAN) latency separating datacenters is orders of magnitude larger than typical network latencies, and this requires a reevaluation of many of the traditional design trade-offs of data management systems. Therefore, data management problems must be revisited to account for the new design space.

In this dissertation, we propose theoretical foundations to understand the limits imposed by WAN latency on GSDM, and propose practical systems and protocols to minimize the overhead caused by WAN latency. The presented work spans global-scale transaction processing, communication, analytics, and machine learning. In all these directions, the focus is on the trade-off between consistency and latency, where we ask the question: *what is the best performance (often latency) we can achieve without compromising the consistency and integrity of data?* For transaction processing, we propose a lower-bound formulation for transaction latency that is imposed by the WAN latency. Also, we propose a new paradigm for transaction processing (proactive coordination) that inspired out two proposed protocols, Message Futures and Helios, which can achieve the lower-bound latency. We also propose a communication framework, called Chariots, to scale multi-datacenter communication. Chariots is carefully designed to allow scaling communication while providing a consistent view of the communicated information.

Finally, we explore challenges in global-scale analytics and machine learning. Specifically, we propose Ogre, a scalable system for global-scale heterogeneous transactional and analytics workloads. Also, we propose COP, a system designed to speed up machine learning on globally generated data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Global-Scale Data Management: Motivation and Challenges

Processing large quantities of data is becoming more ubiquitous and is the driving force behind the sustained growth and impact of Internet Services and Big Data analytics. The way data-intensive applications are deployed has been radically transformed by the cloud computing paradigm realized through massive-scale datacenters. The cloud computing paradigm promises high-performance 24/7 service to users dispersed around the world for cloud applications. Achieving this is threatened by complete datacenter outages and the physical limitations of both the datacenter infrastructure and wide-area communication. To overcome these challenges, systems are increasingly being deployed on multiple datacenters spanning large geographic regions. The replication of data across datacenters (geo-replication) allows requests to be served even in the event of complete datacenter-scale outages. Likewise, distributing the processing and storage across datacenters brings the application closer to users and sources of data, enabling higher

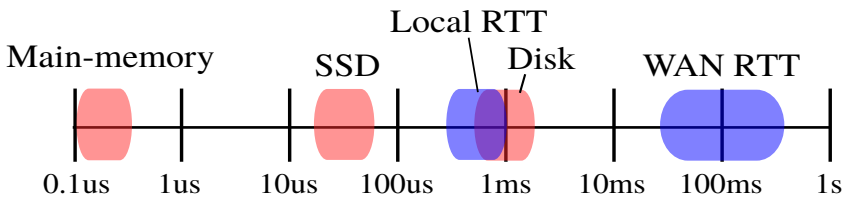


Figure 1.1: Latency of the Wide-Area Network Round-Trip Time communication (WAN RTT) compared to memory access latency [1] and network latency within the datacenter (local RTT)

levels of availability and performance. Additionally, extremely large applications consume huge amounts of resources. These resources vary and include computing infrastructure in addition to power and real-estate. Globally distributing the computing infrastructure of large applications allows them to utilize resources beyond the restrictions of a single datacenter or cloud provider.

Moving to global-scale data management (GSDM), despite its benefits, raises many novel challenges that are not faced by traditional deployments. These challenges include: high wide-area latency, new fault-tolerance model and characteristics, limited wide-area bandwidth, new edge and mobility infrastructure, and regulatory constraints. The large WAN communication latency is orders of magnitude larger than traditional communication latency (See Figure 1.1). This invalidates the traditional space of design trade-offs and makes the WAN latency a significant bottleneck. Traditional database systems are designed with the assumption that the performance bottleneck is dominated by disk I/O, communication bandwidth for distributed databases, and processing power for computationally intensive analytics. With communication latencies that are orders of magnitude higher than other overheads, communication latency between datacenters is the main performance bottleneck in GSDM. This overhead is especially visible for computation tasks that require coordination between various parts of the system that can be across datacenters, such as transaction processing and consistent fresh analytics. This invites

a redesign of data management systems to consider the latency as the main bottleneck. Thus, computation power, memory, disk access, or bandwidth can be traded-off for better latency. In the dissertation, we pursue such a direction and show that by factoring in this new space of trade-offs, GSDM performance is improved significantly.

Fault-tolerance is one of the main motivations of GSDM. However, understanding the nature of datacenter-scale failures is important to build effective fault-tolerance models. This is especially important given that current fault-tolerance mechanisms are built for cluster-based or intra-datacenter environments, which have different failure characteristics compared to datacenter-scale failures. A recent study by Gunawi et. al. [2] gathered information about more than a thousand datacenter failures. We are interested in failures due to power outages and natural disasters. The study shows that on average, 50% of services experience at least three outages annually. This shows that datacenter-scale outages are frequent enough as to be a serious threat to the availability of services. The study also shows that only 3% of these datacenter-scale failures are due to natural disasters (*i.e.*, failures that potentially affect multiple datacenters within an affected region). To tolerate such failures, replication to nearby datacenter is not sufficient—rather replication must utilize distant datacenters. In addition, there are types of failures that can affect a group of datacenters regardless of their locations, such as configuration and software-level failures. These failures are not solved by redundancy (via replication) alone, however, redundancy is an essential component to any solution to such problems.

This understanding of the nature of datacenter-scale failures gives us more intuition on the trade-off between latency and fault-tolerance. The replication latency in geo-replicated systems is dominated by the largest RTT required for its replication. Datacenter-scale outages are a real thread making replication to at least another datacenter necessary, which is a latency of 10s up to 100s of milliseconds. Natural disasters threaten multiple datacenters in a region and thus replicating to a distant region becomes necessary to

tolerate a natural disaster. The latency of such replication is in the order of 100s of milliseconds. The type of tolerated failures differs from one application to another. It is possible that some mission-critical application would require a fault-tolerance level that overcomes multiple, distant datacenter failures, although it is unlikely. Likewise, some other applications may be willing to give up the capability of tolerating natural disasters for performance, but they still replicate to datacenters in close proximity to tolerate individual datacenter failures. In the dissertation, we facilitate such flexibility in managing the trade-off between performance and fault-tolerance in global-scale environments. In particular, we isolate the fault-tolerance component from other concerns, which enables finer control on fault-tolerance requirements.

WAN bandwidth ($\sim$100 Tbps [3]) is larger than traditional networks bandwidth. However, big data applications transfer large volumes of data reported to be in the order of hundreds of TBs per day and projected to be increasing in the future [4]. Also, WAN links are shared by all applications in the datacenters connected by the WAN links, which can be in the order of the tens of thousands and increasing. This increasing demand on WAN bandwidth will lead to a bottleneck in WAN communication. The design of efficient mechanisms that better utilize the WAN links are necessary to avoid limiting the growth of global-scale big data applications. In the dissertation, we tackle this problem in two ways: First, by finding opportunities to avoid WAN communication and serve requests locally. This is possible for analytics and read-only queries, where a consistent state anywhere suffices to answer requests with coordination with other datacenters. Also, we find opportunities to trade-off more communication inside the datacenter (which is cheaper) for less communication between datacenters (which is more expensive). Second, we propose a multi-datacenter communication infrastructure that enables efficient control of message exchange between datacenters.

Another challenge is to efficiently utilize emerging applications, such as Internet of

Things (IoT), edge and mobile applications. These applications generate data at a much faster rate and require high-level insights to be learned and computed from such data. To enable such applications, efficient global-scale systems that handle the communication, analysis, and learning are needed. The dissertation presents two global-scale analytics and learning systems in addition to a global-scale communication framework that can be used to handle the increasing rates of data generation. Also, due to privacy legislation concerns in some parts of the world, there is a direction to limit control on where data is placed and at what level of privacy it is stored [5, 6]. These restrictions affect the design of global-scale data placement and task scheduling. In particular, it is not always feasible to assume control on the infrastructure and data. Thus, providing efficient data management systems is important to optimize the performance even when there is no direct control on the placement of data and nature of the workload.

To summarize, building GSDM systems requires a rethinking of data management problems in light of the new design trade-offs and constraints. Adopting the GSDM model and building efficient global-scale solutions enables higher levels of fault-tolerance, availability, and support for emerging applications. In this dissertation, we build solutions to realize these goals. We focus on the trade-off between consistency and latency for transactional and analytical workloads. These solutions also touch upon the other challenges of GSDM either directly by factoring them in the design choices or indirectly where improving latency and consistency improve other aspects of GSDM systems.

1.2 Dissertation Overview

This section provides an overview of the work proposed in the dissertation. This dissertation aims to provide an understanding of the fundamental challenges of GSDM and the design principles that will help in building efficient GSDM systems. In this

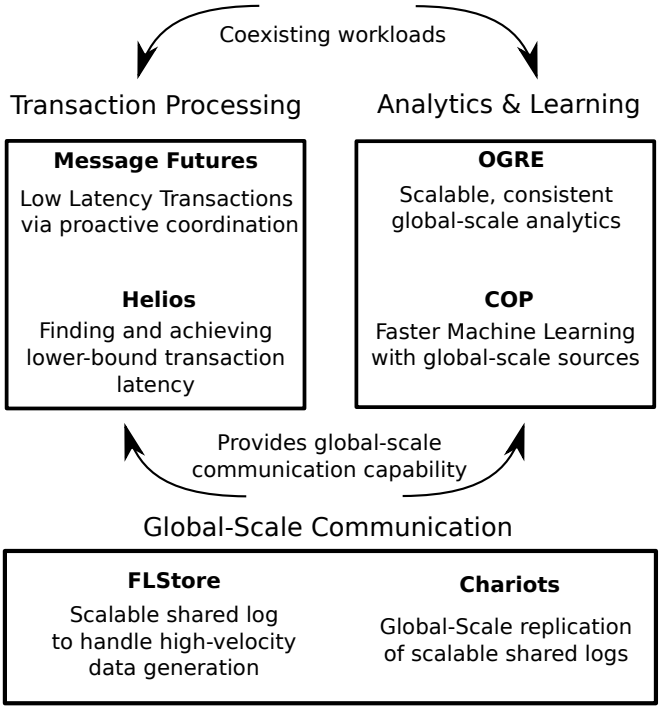


Figure 1.2: An overview of the systems proposed in this dissertation.

process, these design principles are used to build GSDM systems that exhibit them. Figure 1.2 shows the main systems that are proposed in this dissertation. They represent an ecosystem to handle GSDM applications that have workloads that are transactional (*e.g.*, web and cloud applications) and/or workloads to derive insights from data (*e.g.*, Big Data analytics and machine learning). When we tackle building these systems—especially the analytics systems—we consider that transactional and analytical workloads coexist and manage the challenges arising from such coexistence. Additionally, the dissertation tackles a common problem in building both transactional and analytics systems, which is building a global-scale communication framework to facilitate communication between globally-distributed components. Next we overview the individual components of the dissertation.

1.2.1 Transaction Processing

Breaking the Latency Barrier of the Request-Response Paradigm. There is a fundamental coordination latency limit due to the polling nature of traditional protocols that we call the Request-Response paradigm. In the Request-Response paradigm, the coordination for a request starts *after* the request is made, where the replica that received it polls other replicas to inquire about their state and detect conflicts. The request is served only after receiving a *response* from other replicas. This makes a Round-Trip Time (RTT) of communication inevitable—an expensive cost in GSDM. This leads to the question: *Is it possible to avoid the Request-Response paradigm of coordination?* Our work **Message Futures** [7] demonstrates this possibility by an observation that coordination of future requests can start before they arrive. As requests arrive, they are assigned to a predetermined future coordination point. We call this approach *Proactive Coordination*. Coordination points are judiciously calculated to ensure conflicts are detected. A coordination point still needs an RTT for coordination. However, because a request is assigned to a coordination point that already started, the request's observable latency is less than RTT. Message Futures is the first protocol that shows the possibility of faster-than-RTT coordination for all the replicas of a distributed system. Also, it introduces Proactive Coordination, a new approach to coordination that overcomes the limitations of the Request-Response paradigm.

Theoretical Lower-Bound on Coordination Latency. Breaking the RTT latency barrier via the Proactive Coordination paradigm invalidates the previously held convention that coordination cannot be performed faster than the RTT latency. Thus, it opens the question: *What is the lower-bound on coordination latency?* This is a fundamental question in understanding the extent of the effect of the wide-area latency limit on coordination latency. Such a lower-bound, if proven, will provide system designers and

researchers with a theoretical foundation on what is achievable by current and future systems.

To tackle the question of lower-bound coordination latency, We begin by formalizing and modeling the concept of coordination, which is the process of detecting a conflict between two requests. This model of coordination is then used to answer the question: *What are the cases that make detecting a conflict between two requests impossible*? To answer this question, We observe that for any potential conflict between two requests, a and b, one of them must know about the other before making the decision (commit or abort). If they both commit without knowing about the other, then they do not detect the conflict, possibly leading to a consistency violation. My work shows that these cases are inevitable if the coordination latency of a *plus* the coordination latency of b is less than the RTT between the datacenters hosting them. Thus, for any consistent global-scale system, the sum of the coordination latency of any two transactions must be greater than or equal to the RTT between their host datacenters [8]. For example, it is possible that the latency of both a and b is equal to half the RTT between their host datacenters. The coordination model inspired a protocol based on Proactive Coordination, called **Helios** [8] that theoretically achieves the lower-bound, thus proving that the lower-bound is tight. Experimental evaluation shows that Helios approaches the lower-bound in a real global-scale deployment.

The lower-bound result shows that the coordination latency can be faster than what is previously achieved by traditional protocols and even faster than what is achieved by Message Futures. The model of coordination, in addition to being essential for deriving the lower-bound, advances our understanding of the cost of global-scale coordination. It also brings a newfound understanding of the latency characteristics of traditional and Proactive Coordination protocols.

1.2.2 Analytics and Learning

Global-Scale analytics that coexist with transactional workloads. Analytics and maintenance tasks are important for cloud applications. These two classes of workloads have the special feature of being read-only. Our proposed work, Ogre, targets read-only workloads and proposes a transaction commit protocol with special support for read-only transactions while not penalizing coexisting read-modify transactions. Ogre, enables read-only transactions to proceed without any cross-datacenter communication leading to low latency and high throughput. What distinguishes Ogre from traditional read-only transaction protocols is that it is designed for geo-replicated data. Thus, it avoids design patterns that are inefficient for geo-replication like extensive communication between replicas and ordered log shipping. Ogre's design consists of *(1) dependency tracking* to efficiently propagate transactions data between datacenters, and *(2) analytics isolation* that separates the storage units for read-modify transactions and read-only transactions. Unlike recent geo-replicated read-only transaction protocols, Ogre does not require any special hardware support in the form of strictly synchronized clocks. Our evaluation on five datacenters shows that Ogre's read-only transactions execute with low latency and that coexisting read-modify transactions outperform recent geo-replication protocols, even those without special support for read-only transactions.

Machine Learning with Globally-Generated Data. Machine learning is essential for Big Data analytics. This motivated our work on **COP** [9] that specifically targets efficiently supporting global-scale machine learning workloads. In typical global-scale machine learning, data is collected at different locations around the world and then processed at a centralized location. COP targets improving the learning performance for this *Collect then Learn* pattern. COP's main purpose is to pre-process data as it is collected so that when it is received at the centralized location it can be processed

faster. COP's approach ensures a partial order of the execution that will preserve the used machine learning algorithm's theoretical properties. The pre-processing allows COP to enforce the partial order with light-weight operations that outperform traditional methods.

1.2.3 Communication Infrastructure

Global-Scale Data Communication. Large-scale Big Data applications process massive amounts of data that cannot be supported by traditional communication protocols. We address this problem by investigating communication designs that scale to the needs of large-scale applications and be able to support coordination-oriented communication, such as the communication needed for our proposals, Message Futures, Helios, Ogre, COP in addition to other consistent GSDM protocols. **Chariots** [10] is the product of this investigation. To scale Chariots to large-scale workloads, the task of communication is made a priority. Chariots manages a group of machines dedicated for multi-datacenter communication that provides global-scale communication as a service to applications. The problem of communications scalability is tackled by observing that the traditional total ordering guarantees of communication protocols are too strict for coordination-related communication. Rather, it turns out that *causal-order* guarantees are sufficient for coordination-related communication. The design of Chariots proposes novel methods of managing distributed causally-ordered communication that enable it to scale to the demands of large-scale applications.

1.3 Contributions

In this dissertation, we propose an ecosystem of GSDM systems that handles transactional and Big Data analytics workloads at a global-scale. Through this big picture and

tackling the problems of individual components of the GSDM ecosystem, we have gained a fundamental understanding of many of the GSDM challenges and design principles. In all the studies conducted in this dissertation, we start by understanding a fundamental challenge of GSDM, then propose design principles and protocols to tackle that challenge, and finally, we evaluate the proposed solutions on a real global-scale infrastructure. The impact and contribution of the studies presented in the dissertation are presented in the following compact form:

- The dissertation provides an **overview that identifies the opportunities and salient challenges of GSDM.** This has been done via a careful study of the recent literature and practical systems that target GSDM and related fields. We aspire that this overview of GSDM will inspire solving the remaining challenges of GSDM systems and will increase adoption GSDM for various data management fields.

- We propose an **GSDM ecosystem** (see Figure 1.2) that targets a wide-range of data-intensive and Internet applications. This ecosystem supports coexisting transactional, analytics, and learning workloads. The individual components of the ecosystem are systems that support individual workloads. Also, a communication framework is proposed to support the unique global-scale communication needs of the GSDM ecosystem and applications. We envision that this ecosystem can represent a foundation for future GSDM protocols that support data-intensive and Internet applications.

- The dissertation proposes **Proacive coordination** for transaction processing, as an opposed to reactive coordination. In proactive coordination, nodes coordinate with each other continuously for coming and future transactions, rather than starting the coordination of a transaction after it is received. Message Futures [7] and Helios [8] are products of the proactive coordination approach.

- We propose **Message Futures** [7]. Message Futures breaks the Round-Trip Time (RTT) latency barrier for strongly consistent transaction coordination. This is especially important for GSDM, where coordination latency is expensive. Additionally, Message Futures enables dynamically controlling the relative performance (in terms of transaction latency) of datacenters. This enables many useful features for GSDM, such as giving more priority to datacenters at locations where users are active (*i.e.*, following the sun).

- A theoretical formulation of the **lower-bound of transaction latency** [8] is proposed. With this formulation, we achieve an understanding of the fundamental limits imposed by wide-area network latency.

- **Helios** [8], a transaction commit protocol that we propose, is inspired from the lower-bound formulation. The Helios protocol can achieve any transaction latency that is allowed by the lower-bound, including the lower-bound transaction latency. Additionally, Helios separates between the concerns of concurrency control and fault-tolerance, which allows a finer control of fault-tolerance requirements.

- We propose a data management system, **Ogre**, that consists of a transaction processing component and analytical processing component. Ogre enables analytical processing tasks to take place completely within a datacenter while maintaining scalability and consistency of results. This involves a specialized design of the transaction processing components that enables tracking consistency dependencies and enforcing them in the analytical processing component.

- **COP** [9] is our proposal to process machine learning tasks with globally-distributed data sources. COP presents a method to plan execution at data sources that can be used at processing time with the goal of improving performance (in terms of

throughput). The methods proposed in COP can be used for various other tasks and environments where there is an opportunity to plan execution before the actual processing begins.

- We identify global-scale communication as a common challenge that is faced by transactional and analytical systems, in addition to emerging data-intensive applications. To support such systems and applications, we propose **FLStore** and **Chariots** [10] to form a global-scale communication platform for data management systems and applications. FLStore is a scalable shared log storage that is designed to support applications with high data ingestion requirements. Chariots is a system that manages the geo-replication of shared logs across datacenters. Systems and applications can use Chariots and FLStore as a communication layer that is globally-scalable, consistent, and supports large data ingestion rates.

- In all our studies we build prototypes of the systems and protocols we propose. We **evaluate our proposal using these prototypes on real global-scale deployments spanning multiple datacenters.** Such evaluations provide insights on the practical issues faced in GSDM systems in addition to the impact of real-world factors, such as the variability of the underlying resources performance, on performance.

1.4 Organization

The remainder of the dissertation begins with a survey of the related literature on distributed data management and global-scale data management in Chapter 2.

Part I presents our proposed work on global-scale transaction processing. We beging with Message Futures in Chapter 3 and then present the lower-bound formulation and

Helios in Chapter 4.

Part II presentes our proposed work on global-scale analytics and machine learning. Chapter 5 presents Ogre, a transaction management protocol that handles analytics queries efficiently and Chapter 6 presents COP, a framework to execute machine learning tasks with globally-distributed data sources.

Part III presents our work on building a communication infrastructure for GSDM that includes Chapter 7 introduces Chariots, our global-scale communication framework. The dissertation concludes in Part IV that includes a summary of the dissertation in Chapter 8 and a discussion of future directions in Chapter 9.

Chapter 2

Overview of Related Work

In this chapter, we overview the related literature to the topic of global-scale data management. We begin with early databases literature about transaction processing on replicated data. Then, we present recent works that target areas covered by the dissertation.

2.1 Transaction Processing on Replicated Databases

2.1.1 Database Transactions: Definition and Correctness

The goal of databases is to provide easy-to-use abstractions to access data. Database transactions [11], or transactions for short, is a fundamental abstraction of databases and has been under extensive study for the past several decades. A transaction is a group of read and write operations. Databases manages the execution of concurrent transactions with the goal of increasing concurrency without compromising the integrity of data. To preserve the integrity of the data, the database ensures correctness conditions on the outcome of concurrent transactions. *Serializability* is the correctness (also called *isolation*) criterion that emerged with the notion of transactions. It specifies that a group of n

transactions may execute concurrently only if their outcome is equivalent to some serial execution of the n transactions. The reasoning behind serializability is that a programmer writes a transaction to be correct if executed by itself. Thus, an outcome that is equivalent to executing transactions one-by-one is also correct. Since then, there have been many works on isolation criteria for transactions that explore the trade-off between performance and the strictness of an isolation criterion, some of which we will present later in this chapter. In addition to isolation, a database transaction must ensure atomicity (all or nothing execution of a transaction), consistency (a transaction brings the database from one valid state to another), durability (a transaction survives failures). The acronym ACID transaction represents a transaction that guarantees all four properties.

In the dissertation, we focus on the guarantee of isolation. Because isolation requires coordination between concurrent transactions, it is the most affected property by large wide-area network latency. Also, we focus on serializability as a criterion of isolation. This is because serializability provides a strong form of isolation, which makes is easier to use and rids programmers from thinking about concurrency anomalies.

To ensure a serializable execution of transactions, a database coordinates between concurrent transactions using various techniques. It is challenging to reason about the correctness of proposed techniques and whether they indeed ensure serializability. This motivated work on formal methods to prove serializability of transaction commit protocols [12]. These methods model possible outcomes of transaction execution (called execution history) of a given protocol. The model utilizes a graph representation where nodes are executed transactions and edges represent various conflict relations between transactions. The absence of cycles in these graph models denotes a serializable execution. The exact formulation of these graph models depends on the type of database, whether it is centralized, distributed, replicated, uses versioning, etc. We utilize such methods in this dissertation to prove protocols' correctness and we provide the details of constructing

conflict graphs when necessary.

2.1.2 Transactions on Replicated Databases

The problem of providing serializable transactions on replicated databases dates back to the early days of databases research [13, 14]. Fundamental concepts were proposed in early work such as Two-Phase Locking, majority voting, read-write validation, and efficient read-only transactions [15, 16, 17, 18, 19, 20, 21]. The interested reader might refer to [14] for a thorough survey of early work in this area.

Two-Phase Locking (2PL) [16] utilizes locks to detect conflicts between concurrent transactions. A transaction must acquire a read lock on data object x before reading x and a write lock on x before writing x. Concurrent transactions cannot own the same lock at any given time. The locks can be released only after a transaction has acquired all necessary locks and performed all read and write operations. At that point, the transaction is called committed. 2PL ensures correctness because it explicitly orders conflicting transactions as locks do not allow two conflicting transactions to commit at the same time. 2PL can be used to manage transactions on replicated databases, which is our interest for the work in this dissertation. There are different ways to implement 2PL for replicated databases. For example, a primary-copy 2PL [22], requires that all locks are held in the primary copy. Another deployment is to require holding write locks in all copies and read locks in any copy [14].

Thomas [17] proposed the use of voting to commit transactions, where nodes vote whether to commit a transaction. In such an approach, a transaction only needs a majority quorum of positive votes to commit. This was one of the earliest works that utilizes the idea of *validation* (also called *certification*) in what is now called optimistic concurrency control. Kung and Robinson [15] distinguishes between read and write quorums for voting

techniques, which allows assigning flexible quorums. The only restriction is that read and write quorums must intersect. An extension to the distributed case was proposed by constructing a dependency graph to detect conflicts [18], which requires significant computation.

Timestamp ordering [23] is another early approach to coordinate transactions. The main idea is to timestamp transactions and order reads and writes according to the timestamps of their transactions. Different approaches implement timestamp ordering differently, and sometimes it is augmented with 2PL and voting-based approaches.

The interest to prioritize read-only transactions on replicated databases started in [19] where read-only transactions do not have to be validated. However, read-write transactions still need to be validated against read-only transactions. Updates are not installed while conflicting read-only transactions are still in progress, which could lead to significant delay. To overcome this limitation, a multi-version technique was proposed in [20] to execute read-only transactions that do not conflict with read-write transactions. Providing read-only transactions that are served from a single replica was proposed by using gossip and log shipping between datacenters [24]. This approach might be applied to commit protocols that use ordered log shipping for communication [7, 8, 10]. Ordered log shipping, however, creates a bottleneck and limits scalability. For this reason, Ogre adopts a dependency tracking scheme to enable read-only transactions. Dependency tracking is more scalable than ordered log shipping because it only enforces necessary transaction dependencies rather than the log's total order. Spanner [25] supports local read-only transactions by guaranteeing external consistency via clock synchronization that is enabled by a specialized infrastructure that leverages atomic clocks and GPS, which is not accessible to most systems. In contrast, Ogre avoids the use of time synchronization (including loose synchronization) as a basis for concurrency control.

2.2 Paxos Consensus Protocol and Variants

Paxos [26, 27] is a consensus protocol that is widely used to implement replicated databases as Replicated State Machines (RSM). Paxos provides a mechanism to agree, among nodes, on the execution order of commands (*e.g.*, transactions). Due to its wide-use in recent cloud and global-scale databases, we provide an overview of Paxos, its variants, and usage for GSDM.

A consensus algorithm solves the problem of deciding (also called choosing) a single value among proposals by multiple nodes. Deciding a value in Paxos is done in two phases: a *leader election* phase and a *replication* phase. When a node receives a request to propose a value, it attempts to become a leader by starting a *leader election* phase. If the node receives a majority positive *promises* in this phase, it becomes a leader. Then, it starts a *replication* phase where it proposes its value. If the proposed value is *accepted* by a majority, then the value is said to be *decided*.

Paxos resolves concurrent proposals using unique *proposal ids*. In the leader election phase, a proposal is stamped with a unique id, p, and a prepare(p) message is sent. Assume that node A sent the prepare message. Processes receiving the prepare reply with a promise() message if p has the highest proposal id from ones they already received. With the promise, the most recent accepted proposal q, if any, with its value v_q are also sent. Process A is considered to be elected a leader if it receives a majority of promises for p. Then, A proceeds to the replication phase with the proposal p', which corresponds to the highest received proposal, q. Otherwise, A proceeds with its own proposal if no proposals were sent along with promise() messages. In the replication phase, a value v associated with p' is sent in a propose(p',v) message. A node replies with an accept(p') message if the proposal id is greater than or equal to the highest promised proposal. Once A receives a majority of accept(p') messages, then the value v is decided. If any step fails along this

algorithm, a node might retry again with a higher proposal number. If a value is already decided, the node might try to compete for higher slots.

In practice, Paxos is used to decide a sequence of values, where each position is called a *slot*. Multi-Paxos [27] is an efficient variant of Paxos that optimizes for this case. Rather than performing the leader election phase for all slots, an elected leader for a slot i maintains the role for future slots and is called a *prolonged leader*. Requests for slots after i would be directed to the prolonged leader, and the prolonged leader bypasses the leader election phase, thus reducing the number of needed phases from two to one.

Fast Paxos [28] optimizes the number of message delays by sending requests directly to all replicas rather than sending them to the prolonged leader first. This enables deciding values in one round rather than two for some cases. However, the required quorum for Fast Paxos is larger than a majority. Thus, in edge data management, Fast Paxos might require communicating with more zones than what is required by a majority. Generalized Paxos [29] allows values that are not conflicting with each other to be decided concurrently, thus improving throughput. MDCC [30] shows an efficient use of both Generalized and Fast Paxos to commit transactions on geo-replicated data. Flexible Paxos [31] proves the feasibility of having a quorum allocation other than a majority or super majority for the leader election and replication rounds. They demonstrate this by integrating different quorums allocations in Paxos such as grid quorums.

There have been proposals for variants of the Paxos protocol specifically to global-scale data management and transaction processing. Egalitarian Paxos (EPaxos) [32] is a Paxos variant that decides values in a single round without a designated leader. In the case of a conflict, another round is needed. Conflicts are resolved by tracking dependencies between conflicting values. Paxos-CP [33] proposes the use of two optimizations to improve the concurrency of Multi-Paxos: (1) *Combination*: Two values may be concurrently decided if they correspond to two transactions that do not conflict with each other, and (2)

Promotion: A losing value in a slot i is immediately promoted to compete in slot $i+1$ if it does not conflict with the decided value in slot i. These two optimizations save the processing time of reevaluating the request to come up with a new value if it does not conflict with concurrent values.

Other Paxos variants tackle the problem of load balancing. Because Paxos is a leader-based approach, leaders are more utilized than the other parts of the system. In a global-scale setting, this translates to having more load at the datacenters with leaders. Mencius [34, 35] is a Paxos-based protocol that targets achieving efficient load balancing by rotating the leader for each slot. Another Paxos-based protocol that tackles load balancing is S-Paxos [36] that offloads some of the work of the leader to all datacenters.

Some systems that are based on Paxos, such as Megastore [37], distinguish between voting and non-voting (read-only) replicas. Thus, the read availability increases with additional read-only replicas without affecting the performance of updates at voting replicas. Read-only replicas cannot replace the need for globally-distributed voting replicas. This is because a replica can only be a leader if it is a voting replica. Thus, globally-distributed voting replicas are needed in scenarios where leaders must be able to exist in geographically separated locations.

2.3 Global-Scale Transaction Processing

2.3.1 Eventual Consistency and Key-Value Stores

The central problem of data access to globally-distributed data is managing the consistency-performance trade-off that is amplified due to large wide-area latency. To guarantee consistent outcomes, coordination between distributed components is necessary. This coordination is expensive due to WAN links, and thus affects performance immensely.

The first reaction to the new trade-off was the abandonment of easy-to-use abstractions and isolation guarantees in favor of performance [38, 39, 40]. Systems that adopt this approach provide weak guarantees like eventual consistency and single-key atomicity. These guarantees were sufficient for many applications; however, it turns out that many use-cases require stronger notions of consistency. Besides, developing applications on top of weakly consistent data is error-prone and less natural to developers.

2.3.2 Relaxed Transactional Semantics

The need for stronger forms of consistency on globally-distributed data sparked a trend to explore different points in the consistency-performance trade-off spectrum. The goal is to explore existing consistency notions, that are weaker than expensive strong consistency, and extend them in ways suitable for globally-distributed data. Causal consistency, inspired by the causal ordering principle [41], preserves causal relations between operations. Causal consistency is attractive for GSDM because it does not require coordination between replicas. Thus, causally consistent systems that are built for globally-distributed data, do not suffer significantly from the cost of wide-area latency [42, 43, 44, 10, 45]. For example, COPS and Eiger [42, 44] are scalable causally consistent systems that offer single-key operations in addition to read-only or write-only transactional access. COPS extends the notion of causal consistency and proposes Causal+ consistency that, in addition to causality, guarantees data convergence.

Snapshot Isolation (SI) [46] guarantees that transactions always read a consistent snapshot and ensures the absence of write-write conflicts. SI can lead to better performance compared to stronger notions of consistency [46]. Many solutions leverage SI for GSDM [47, 48, 49, 50, 51]. Walter [47] extends the notion of SI and proposes Parallel SI (PSI). PSI redefines snapshot reads and write-write conflicts to accommodate the new environment of

globally-distributed data. With PSI, Walter can replicate data asynchronously while still providing strong guarantees within each site. In the original SI, asynchronous replication is not possible.

2.3.3 Strongly Consistent Transactions

Strong consistency guarantees, such as serializability [12] and linearizability [52], require extensive coordination between globally-distributed data. Despite their performance implications, industry and academia have recently shown that strong guarantees are needed for a wide range of applications. This has started a movement towards preserving strong consistency guarantees while trying to reduce the cost of coordination on performance. Paxos [26] is a fault-tolerant consensus protocol that has been used to implement GSDM systems that provide strong guarantees, such as megastore [37], Paxos-CP [33], and MDCC [30]. Later, though, Paxos was shown to perform better as a synchronous communication layer integrated with a transaction commit protocol [25, 53, 54]. For example, Spanner [25] and Replicated Commit [54] commit transactions using variants of Two-Phase Commit (2PC) and Strict Two-Phase Locking (S2PL) and leverage Paxos to replicate across datacenters. Paxos, however, requires two rounds of communication to commit, which is a cost amplified by WAN latency. MDCC [30, 55] proposes the use of Fast Paxos [28] that allows committing with a single round to a super majority rather than two rounds.

The use of timestamps and time synchronization has been explored for GSDM systems [25, 51, 45]. Spanner [25] provides external consistency guarantees by leveraging accurate time synchronization using specialized infrastructure such as atomic clocks. Without accurate time synchronization, other systems resort to loosely-synchronized clocks methods [51, 45].

2.3.4 Coordination-Free Processing

To commit arbitrary transactions with strong consistency guarantees, the cost of coordination is inevitable [8]. However, coordination-free execution is possible for some types of transactions [56, 57, 58]. This is possible by inferring application-level invariants of transactions correctness and then exploring whether a coordination-free execution is permissible. Transaction Chains [58] derives an execution plan of distributed transactions that allows for a fast response time. A client needs to wait for the execution of the transaction at only a single site, rather than all accessed sites. Often, the first site is local to the client, making the response latency unaffected by the WAN latency.

2.4 Global-Scale Data Management

2.4.1 Data Placement

Placement of data and workers has a significant effect on GSDM systems performance [59, 60, 4, 61, 62, 63, 64, 65]. SPANStore [60] proposes an optimization formulation of placement to minimize monetary cost with constraints on fault-tolerance, consistency requirements, and performance SLOs. DB-RISK [59] is an interactive game that invites participants to experiment with placement decisions and optimizations of GSDM systems. Sharov et al. [65] propose an optimization formulation for placement with an objective of minimizing latency. What distinguishes this work is that it considers transactional access to storage.

2.4.2 Managing the Consistency-Performance Trade-off

The dynamic and variable nature of workload and communication links in GSDM led to work on dynamic techniques for GSDM systems [66, 67, 68]. Variability of the commu-

nication link leads to the reordering and delaying of sent messages. CosTLO [66] proposes adding redundancy in messaging to lower the latency variance. GSDM systems trade-off consistency and performance and could be locked in their initial protocol choice. Pileus and Tuba [67, 68] allow applications to dynamically control the consistency-performance trade-off by declaring their consistency and latency priorities. The application's priorities then influence the decision on which servers to access. Stronger consistency requires accessing more servers and thus increases latency. In turn, more relaxed consistency requires accessing lesser number of servers and thus decreases latency.

2.5 Global-Scale Data Processing and Analytics

2.5.1 Events Processing

Global-scale applications receive and generate large volumes of data across datacenters. It is reported that the amount of data processed by large web applications is in the order of 10s to 100s of TBs per day [4]. This has led to the design of many GSDM systems for stream processing and data pipelining from Google [69, 70, 71, 72, 73], Twitter [74, 75], Facebook [76], and LinkedIn [77]. These systems are designed to support the high volume and velocity of events while conserving availability and high performance. Also, they guarantee the correctness of computations that use the streamed data. Correctness invariants vary depending on the application, but the following correctness conditions are common: (1) *At-most-once semantics*, which means that no event is processed more than once, and (2) *Near-exact semantics*, which means that with no significant delay, all events will be processed. These correctness conditions, albeit simple, are challenging on the scale of global web applications where streams might be significantly reordered and delayed. Photon [70] is a system deployed at Google to join global-scale continuous streams. To

tolerate failures, an event can be processed at any of the operating datacenters. To guarantee at-most-once semantics, before a joiner starts processing an event it ensures that the event has not been processed before. It does so by using a logically centralized Paxos process for the event's unique ID.

2.5.2 Machine Learning and Data Analytics

In addition to processing streams and managing data pipelines, GSDM systems often perform analytics and machine learning queries on data to extract insight and business knowledge. Global-scale analytics — as opposed to traditional analytics within a datacenter — face novel challenges. Until recently, global-scale analytics were performed by pulling all data to a central location [78]. This, however, consumes the WAN links, which causes monetary losses and poses a physical constraint on throughput. Additionally, data movement could be constrained due to emerging data sovereignty legislation and privacy concerns. Recent solutions address the challenges of global-scale analytics [78, 63, 4, 79, 80, 81, 82, 83, 84, 85, 86]. Geode [4] and PIXIDA [85] are two systems that propose a query planning and replication framework that targets reducing the bandwidth cost between datacenters. Additionally, Geode uses optimizations such as aggressive caching and measurement collection that allow for the reuse of past queries. Unlike Geode and PIXIDA, Iridium [63] aims to minimize the latency of global analytics by deriving the placement of both data and tasks. Geode, PIXIDA, and Iridium derive their solutions using an optimization formulation. They all face a common problem of the intractability of optimization solvers with 10s of datacenters. To overcome this, Geode and Iridium leverage a greedy heuristic and PIXIDA proposes a flow-based approximation algorithm.

Part I

Low-Latency Global-Scale Transaction Processing

Chapter 3

Message Futures: Proactive Coordination for Transaction Processing

3.1 Overview

In this chapter, we propose Message Futures (MF), a completely distributed multi-datacenter transaction management system that provides strong consistency guarantees while maintaining low commit latency. It achieves an average commit latency of around one Round-Trup Time (RTT). A transaction is committed when a *commit condition* on mutual information is met. At any point in time, the commit condition is designed to be true for any single object in at most one datacenter. A Replicated Log (RLog) [87] is utilized to share transactions and state information among datacenters continuously. RLogs provide enough information to ensure correct transaction execution. Thus, the continuous exchange of RLogs *allows a datacenter to commit transactions without initiating a new wide-area message exchange with other datacenters.* Message Futures guarantees one-

copy serializability [12]. Message Futures can be modified to achieve different degrees of consistency by incorporating suitable reading and conflict detection strategies. Moreover, our design allows easy incorporation of state-of-the-art techniques to allow for more concurrency, such as causal+ consistency [42], commutative updates, and counting sets.

In addition to maintaining strong consistency and low latency, the incorporation of RLog improves Message Futures' resilience to node and communication failures. RLogs guarantee the *happens-before* relation among events. RLogs are propagated between datacenters continuously and independently. We study the propagation of RLogs to achieve the desired performance properties. In particular, we study how assigning different RLog transmission rates allows us to prioritize datacenters and tailor commit latency. This, in many common cases, enables high priority datacenters to commit transactions immediately while having a minimal effect on other datacenters commit latency. We focus on *fixed frequency* propagation, where each node sends RLogs to other datacenters with a fixed frequency. This study can be extended to arbitrary propagation of RLogs (Message Futures correctness is maintained with arbitrary RLogs propagation patterns.)

Our contributions are summarized as the following:

- We propose Message Futures, a concurrency control manager (Section 3.3). It provides strong consistency guarantees while achieving a commit latency close to a single RTT.
- Correctness of Message Futures is proved (Section 3.3.4).
- We performed a set of experiments on AWS EC2 nodes across inter-continental datacenters (Section 3.4). In those experiments, we compare Message Futures performance with two Paxos-based protocols. We also demonstrate how to tailor the system's commit latency and how to enable transactions to commit immediately.

The rest of this chapter is divided as follows: First, we overview the system in Section 3.2 and elucidate the considered datacenter architecture, the main components of an Message Futures instance, and RLogs. Then, we proceed to propose Message Futures in Section 3.3. We provide an intuitive description followed by a more formal definition of Message Futures operation and a proof of its correctness. An evaluation of our system is detailed in Section 3.4. Finally, we conclude with a summary in Section 3.5.

3.2 System overview

We consider a large multi-cloud system consisting of multiple datacenters, each with a large number of nodes. In what follows we describe the underlying datacenter architecture, Message Futures design, and RLogs.

3.2.1 Datacenter architecture

Each datacenter contains a subset of the nodes that belong to the system, referred to as a *cluster*. Each cluster maintains a full replica of the system's data. Also, they, or a subset of them, receive and answer clients' requests. We treat the datacenter as a three-tiered architecture. The *application tier* is responsible for handling clients' requests. Concurrency and replication are handled in the *transaction tier*. Storage and maintenance of data are the responsibility of the *storage tier*. A key-value store constitutes the underlying storage. The storage can be distributed across different servers in the cluster. Since latency between servers within a single datacenter is small, applying current solutions for intra-datacenter concurrency does not affect the overall performance of a geo-replicated system. Our solution handles inter-datacenter concurrency control. Message Futures assumes that transactions executed within a datacenter are serializable. A complete datacenter outage is a rare occurrence. Message Futures focuses on optimizing the normal

case operation, that is when all datacenters are available. Clock synchronization between datacenters is not required by Message Futures.

3.2.2 Message Futures design

Message Futures is a multi-datacenter concurrency control manager. Components in Message Futures' design include: (1) *The transaction client component* to handle clients requests. (2) *The replication component* to handle RLog replication. (3) *The concurrency component* that performs the concurrency logic to commit or abort transactions. These components operate on common global data structures. Each datacenter, DC_i, maintains the following structures:

- *Local RLog*, L_i, is the locally maintained RLog.
- *Pending Transactions list*, PT_i, contains local *pending transactions*. These are transactions that requested to commit but are still neither committed nor aborted.
- *Last Propagated Time*, LPT_i, is the timestamp of the processing time of the last sent L_i at DC_i.

3.2.3 Replicated Logs

Efficient propagation of the system's information can be achieved with the use of *RLog*. RLogs maintain a global view of the system that can be used by datacenters to perform their concurrency logic. RLogs consist of an ordered sequence of *events*. All events have timestamps. Each transaction is represented by an event. RLogs are continuously propagated to other datacenters. By constantly propagating RLogs to other datacenters, a full knowledge of the system's state is maintained by all datacenters. An algorithm used to efficiently propagate RLogs is presented in [87]. An $N \times N$ Timetable, T_i, is

maintained by L_i, where N is the number of datacenters. Each entry in the Timetable is a timestamp representing a bound on how much a datacenter knows about another datacenter's events. For example, entry $T_i(j,k) = \tau$ means that datacenter DC_i knows that datacenter DC_j is aware of all events at datacenter DC_k up to timestamp τ. An event in L_i is discarded if DC_i knows that all datacenters know about it. RLogs are transitive: events of a datacenter DC_A may reach DC_C through another datacenter DC_B. The algorithm ensures two properties. First, all events are eventually known by all datacenters. Second, if two events have a *happened-before* relation [41], their order is maintained in the RLog.

Now, we will illustrate our adaptation of the RLog. Each datacenter is represented by one row and one column in the Timetable. Each transaction, t_i, is represented as an event record, $E_{type}(t_i)$, in the RLog, where *type* is either: (1) *Pending transactions* (p) that requested to commit but are neither committed nor aborted yet. (2) *Committed transactions* (c) that were not propagated to all datacenters yet. A transaction, t_i, starts as a pending transaction with an entry record $E_p(t_i)$. Once the transaction commits or aborts, a new event record, $E_c(t_i)$, is added to the RLog. A pending event is maintained until the transaction commits or aborts. A committed event is maintained in the RLog until it is known to all datacenters. Note also that the clock must be incremented when new events occur and when L_i is propagated.

3.3 Concurrency control

In this section, we describe the concurrency control manager. The section begins with an overview and an example scenario of Message Futures. Then, Message Futures' algorithms used to commit transactions are presented. Finally, correctness proofs are presented.

3.3.1 Commit protocol overview

We will now give an intuitive description of our commit protocol. There are two main properties that are enforced. The first property is keeping implicit promises. When a site transmits a replicated log, it contains information on committed transactions (we will mention pending ones shortly). Obviously, those are ones committed prior to the transmission. The property is that a site, S_i, upon sending the log, at a time τ, promises all other sites that it wont commit any other transaction until it knows that other sites heard that promise. This is reflected by the following condition: $T_i(j, i)$ for any site S_j is greater than or equal to Γ, where Γ is the last time the local log was processed prior to τ. The second property is that a site, S_i, commits a transaction only if all other sites have active implicit promises while S_i have no promises to any other site.

So far we considered promises to not commit transactions, no matter what is its read- and write-sets. Also, a promise is valid for one round of log exchange[1]. This is a very restrictive case. To overcome this restriction, information of pending transaction are also included. Promises in this case become promises of not committing any transaction, *except* for those pending ones. We call them *less restricted promises*. Thus, other sites can compare against those pending transactions and commit transactions that do not conflict with it and satisfy the second property with the less restricted promise. Also, by persisting the entry of the pending transaction until it commits/aborts, the less restricted promise can span multiple rounds of log exchange.

3.3.2 Message Futures overview

We start with an intuitive description of Message Futures. Note that each datacenter, DC_A, transmits L_A, its local RLog, continuously regardless of the existence of new events.

[1] a round of log exchange is a transmission of the log from site A to B then the transmission of a log from site B to A.

Consider a pending transaction t_i at DC_A. When t_i requests to commit, the current Last Propagated Time, LPT_A, is attached to t_i and is referred to as $t_i{\rightarrow}LPT_A$. Then, t_i with its read- and write-sets are appended to the local Pending Transactions list, PT_A, while only the write-set is appended to L_A. Whenever DC_A receives a RLog, L_B, it checks for conflicts between transactions, t_i, in PT_A and t' in L_B. If a conflict exists, t_i is aborted. A conflict exist if a common object, x, exists in t''s write-set and t_i's read- or write-sets. To commit t_i, DC_A waits until the following *commit condition* holds:

Definition 1 *A pending transaction t_i in PT_A commits if all read versions of objects in t_i's read-set are identical to ones in local storage, and*

$$T_A[B, A] \geq t_i{\rightarrow}LPT_A, \quad \forall_B(DC_B \in datacenters)$$

That is, all objects in t_i's read-set have the same versions as those in the local storage and datacenter DC_A knows that all datacenters, DC_B, are aware of DC_A's events up to time $t_i{\rightarrow}LPT_A$. Conflicts that include t_i's write-set are detected earlier when remote transactions are received and their conflicts are detected.

To illustrate the operational aspects of Message Futures and its correctness, consider a successfully committed transaction, t_i, at datacenter DC_A as an example. The event of transmitting L_A by DC_A is denoted by a_α, where α is a monotonically increasing number; a_α is before a_β if $\alpha < \beta$. The time of an event is represented by the notation $ts(e)$, where e is the event and the returned time is a locally maintained monotonically increasing number. If transaction t_i requests to commit at datacenter DC_A at time $ts(t_i)$, where $ts(a_\alpha) < ts(t_i) < ts(a_{\alpha+1})$, then $t_i{\rightarrow}LPT_A$ is set to $ts(a_\alpha)$. This ensures that the event record of t_i will be included in L_A of every a_β, where β is greater than α. The transaction commits if the commit condition holds. That means the following: (1) No conflicts were detected between t_i and transactions in L_A until now, and (2) All other datacenters, DC_B, have acknowledged knowing about DC_A's transactions prior to $ts(a_1)$.

For any other datacenter, DC_B, consider b_α as the first transmission of L_B, where L_B's Timetable satisfies: $T_B[B, A] \geq ts(a_\alpha)$. This is DC_B's part in satisfying t_i's commit condition. Reception of b_α's content by DC_A makes it aware of all transactions, t_{b_pre}, with commit request times less than $ts(b_\alpha)$. No t_{b_pre} conflicts with t_i; otherwise, t_i would have been aborted, since t_{b_pre} is necessarily included in L_B which is transmitted in b_α. Any other transaction, t_{b_post}, that requested to commit after $ts(b_\alpha)$, is necessarily aborted if it conflicts with t_i. This is because $t_{b\_post} \rightarrow LPT_B$ is greater than or equal to $ts(b_\alpha)$; t_{b_post} will wait until its commit condition holds. This includes waiting for $T_B[A, B]$ to be greater than or equal to $ts(b_\alpha)$. $ts(b_\alpha)$ is greater than the time when a_α's transmission was received by DC_B. Thus, any L_A satisfying t_{b_post}'s commit condition must have been sent at a time greater than or equal to $ts(a_{\alpha+1})$. L_A sent at a time greater than or equal to $ts(a_{\alpha+1})$ must contain t_i, as we have shown above. Thus, when the record of t_i is received at DC_B, t_{b_post} is aborted if a conflict existed between t_i and t_{b_post}. Since any transaction in DC_B requests to commit either before or after $ts(b_\alpha)$, the commit condition ensures that for any two transactions that are concurrent and conflicting, at most one of them will commit.

We now illustrate a simple operational scenario of Message Futures depicted in Figure 3.1. The scenario consists of two datacenters, DC_A and DC_B. The passage of time is represented by going downward. Arrows are RLog transmissions. Events in the RLog are shown over the arrow. If no events exist, nothing will be shown. The corresponding Timetable is also displayed in one case for demonstration purposes. The notation on the sides are operations performed or issued at the datacenter. $t_i.operation(key)$ represents performing an operation on the object *key* for transaction t_i. Client operations are *read* (r), *write* (w), and *commit request* (cr). Commits and aborts are shown inside dotted boxes. As introduced above, RLog transmissions are represented by the notation δ_i, where δ is the lower case character of the datacenter's name and i is a monotonically increasing

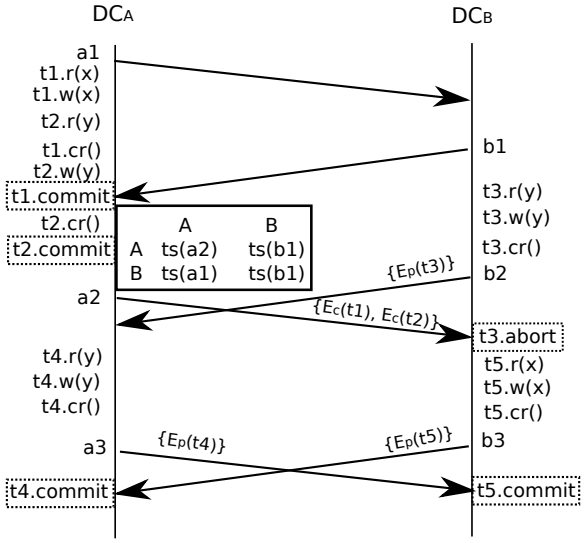


Figure 3.1: Message Futures example scenario

number.

Consider transaction t_1 of DC_A. It reads and writes object x and then requests a commit. $t_1 \rightarrow LPT_A$ is set to $ts(a_1)$. DC_A waits until the commit condition (Definition 1) holds. When L_B, sent at b_1, is received at DC_A, the commit condition is satisfied and t_1 commits. Transaction t_2, which also started after a_1, requests a commit. $t_2 \rightarrow LPT_A$ is also set to $ts(a_1)$. Since it has requested to commit after the reception of the RLog transmission at $ts(b_1)$, the commit condition holds at the time it requested to commit, hence t_2 commits immediately. Transaction t_3 requests to commit at DC_B. $t_3 \rightarrow LPT_B$ is set to $ts(b_1)$ when a commit is requested. However, when L_A of a_2 arrives at DC_B, a conflict with transaction t_2 is detected. In this case, t_3 is aborted. Finally, we show the case of transactions t_4 and t_5. When a commit is requested for both of them, $t_4 \rightarrow LPT_A$ is set to $ts(a_2)$ and $t_5 \rightarrow LPT_B$ is set to $ts(b_2)$. When each datacenter receives the other datacenter's RLog, it contains the information of the pending transaction of the other datacenter. However, no conflict is detected. At that point, the commit condition holds for both of them and both t_4 and t_5 commit. We also included a demonstration of T_A at

time $ts(a_2)$.

3.3.3 Concurrency control protocol

The concurrency component of a datacenter DC_A runs continuously to process L_A and PT_A. A client can issue four types of operations on a transaction t. A *Begin* initializes t's record and acquire a locally unique identifier. *Read* operations add the key, read value, and version to the read-set and returns the value to the client. $Write$ operations add the key and written value to the write-set. Finally, Clients read local values and buffer write operations. Conflicts with the local storage are checked by verifying that all read values of t_i have the same version number as the last writes on the local storage. Also, conflicts with other transactions in PT_A, pt, are checked. If a conflict is detected between t_i and pt, t_i is aborted. A conflict exists if a common object, x, exists in pt's write-set and t_i's read- or write-sets. When a transaction, t_i, requests to commit at DC_A, it is added to PT_A (with both read- and write-sets) and L_A (with its write-set only) and $t_i \rightarrow LPT_A$ will be set to the current LPT_A. After each RLog transmission, LPT_A is set to the processing time of L_A, which is the entry $T_A[A, A]$. Upon receiving a RLog, L_B, from any DC_B, L_A is updated to reflect the received information, *i.e.*, L_B's events are merged into L_A's events and T_A is updated.

A concurrency server, at DC_A, runs continuously and at each iteration performs three tasks in the following order:

Process transactions, lt, in L_A: This step is demonstrated in Algorithm 1. Two types of transactions in L_A are considered: (1) Committed transactions that were not processed before, and (2) pending transactions that did not commit/abort yet. Local transactions in L_A are not considered (line 3). Other transactions are processed according to their order in L_A and checked for conflicts with transactions, pt, in PT_A (lines 5-7). If

Algorithm 1: Processing transactions in L_A.

```
1  function ProcessLocalLog()
2      for each transaction lt in L_A do
3          if (lt was already processed AND is not pending) OR (lt is local) then
4              continue to next lt
5          for each transaction pt in PT_A do
6              if conflictExist (pt, lt) then
7                  abort pt
8          if lt is committed then
9              apply lt's write − set to storage
```

Algorithm 2: Processing transactions in PT_A.

```
10 function ProcessPendingTransactions ()
11     for each transaction pt in PT_A do
12         if for all datacenters, DC_B, T_A[B, A] ≥ pt→LPT_A AND read-set values are not
           changed in local storage then
13             apply pt's write − set to storage
14             add E_c(pt) to L_A
15             remove pt from PT_A
```

a conflict was detected between lt and pt, then pt aborts. A conflict exist if a common object, x, exists in lt's write-set and pt's read- or write-sets. Finally, if lt is a committed transaction, incorporate its write-set to the local storage (line 9).

Process transactions, pt, in PT_A: Determine if any pt can commit (Algorithm 2). pt can commit if the commit condition holds (line 12). Committing a transaction includes incorporating its write-set to local storage, adding a committed event record, $E_c(pt)$, to L_A, and removing pt from PT_A (lines 13-15).

Garbage collection and termination: If a transaction, t_i is committed/aborted then discard $E_p(t_i)$. If t_i is committed/aborted and is also known to all datacenters, then discard $E_c(t_i)$.

3.3.4 Correctness

We now show that Message Futures preserves one-copy serializability. We use Serialization Graphs (SGs) [12]. Nodes in a SG represent committed transactions and edges represent conflicts. An edge (t_i, t_j) represents one of the following three types: (1) *Write-read (wr) edges* where t_j directly *read-depends* on t_i. (2) *Write-write (ww) edges* where t_j directly *write-depends* on t_i. (3) *Read-write (rw) edges* where t_j directly *anti-depends* on t_i. A Multi-Version, Multi-Copy (MVMC) history is one-copy serializable if no cycles exist in its SG. Acyclicity of SG will be proven by showing that any edge (t_i, t_j) translates to an ordering of the commit record of t_i, $E_c(t_i)$, preceding ($<_s$) $E_c(t_j)$ in RLog. Since the order of events in RLogs maintains the *happens-before* relation, and from the acyclicity of the *happens-before* relation, proving this mapping establishes the acyclicity of SG. This is an intuitive mapping since the ordered list of events in RLog actually represents a totally ordered history of committed transactions. We begin by listing some properties of Message Futures to aid us in proving this mapping.

Property 1 *For a transaction t_j at DC_A, define the set of previously committed transactions by TX_p, where*

$\forall_{t \in TX_p}$ (write-set(t) $\cap$ read-set(t_j) $\neq \phi$) *OR* (write-set(t) $\cap$ write-set(t_j) $\neq \phi$). *All transactions in TX_p have event records in L_A at the time of t_j's commit.*

The property asserts that prior to transaction t_j's commit, DC_A accumulates the history of all previously committed transactions with at least one object x in their write-set that is also in t_j's read- or write-sets.

Property 2 *The set of committed transactions, TX_c, that were concurrent at any point in time satisfies the following:*

$\forall_{t_j \neq t_i \in TX_c}$ (write-set(t_j) $\cap$ read-set(t_i) = write-set(t_j) $\cap$ write-set(t_i) = ϕ).

This property states that concurrent transactions can commit only if they have disjoint object sets. Now, we will consider each edge type and prove our mapping:

Write-read edges. All transactions read from their local storage. Consider the wr-edge (i, j). A transaction t_j at datacenter DC_A contains version x_v in its read-set. Property 2 indicates that t_i and t_j are not concurrent, thus t_i committed before t_j's commit request. Given Property 1 we also know that all transactions that have key x in their write-sets are in L_A. This includes transaction t_i that wrote the most recent version, *i.e.*, x_v. Thus, the order $E_c(t_i) <_s E_c(t_j)$ is guaranteed in L_A and there is no transaction t_m that writes to x having an order $E_c(t_i) <_s E_c(t_m) <_s E_c(t_j)$. Since the RLog maintains the order of the *happens-before* relation, $E_c(t_i) <_s E_c(t_j)$ will hold for all RLogs.

Write-write edges. Consider two transactions, t_i and t_j, with a common object, x, in their write-sets. Consider an edge (i, j) in SG. We infer from Property 2 that t_i and t_j are not concurrent; t_i committed before t_j requests a commit. Furthermore, Property 1 indicates that L_A accumulated the set of all previously committed transactions, TX_p, that have at least one common object in their write-sets with t_j. Thus, t_i which wrote the most recent version of x before t_j is necessarily in TX_p. Thus, the order $E_c(t_i) <_s E_c(t_j)$ is ensured in L_A and there is no transaction t_m that writes to x with the order $E_c(t_i) <_s E_c(t_m) <_s E_c(t_j)$. The order holds for all RLogs due to the preservation of the *happens-before* relation.

Read-write edges. Consider the case of three transactions, t_i, t_j, and t_k accessing an object x. Assume that t_j and t_i request to commit at DC_A and DC_B respectively. Assume there is a ww-edge from t_k to t_j and a wr-edge from t_k to t_i, then there should be a rw-edge from t_i to t_j. We need to prove the mapping of those transactions to an order $E_c(t_k) <_s E_c(t_i) <_s E_c(t_j)$. We already proved that $E_c(t_k) <_s E_c(t_i)$ and $E_c(t_k) <_s E_c(t_j)$. We now need to only prove that $E_c(t_i) <_s E_c(t_j)$. Suppose to the contrary that $E_c(t_j) <_s E_c(t_i)$. This is only possible if $E_c(t_j)$ was already in L_B when t_i

commits. There are three cases of possible times for t_j's reception at DC_B: (1) $E_c(t_j)$ was in L_B before t_i requested to commit. This case is not possible, since Message Futures compares the read versions against the ones in storage, and having $E_c(t_j)$ in L_B necessarily means that t_j's write-set was incorporated in DC_B. (2) $E_c(t_j)$ was added to L_B while t_i was a pending transaction in PT_B. This is also not possible because all pending transactions are checked for conflicts with any externally received transaction. So, when $E_c(t_j)$ was first received at DC_B, transaction t_i would have been aborted. (3) $E_c(t_j)$ arrived at DC_B after t_i commits. However, when t_i commits, the entry $E_c(t_i)$ is added to L_B. Thus, $E_c(t_j)$ is necessarily added after $E_c(t_i)$, hence $E_c(t_i) <_s E_c(t_j)$. A contradiction is observed for all cases. Having no transaction t_m that writes to x with an order $E_c(t_i) <_s E_c(t_m) <_s E_c(t_j)$ follows from our proof of ww-edges.

The following theorem shows that our mapping is sufficient to prove one-copy serializability.

Theorem 1 *Message Futures ensures one-copy serializability.*

Proof: Message Futures guarantees the mapping of SG edges, (i, j), to an order $E_c(t_i) <_s E_c(t_j)$ in RLogs as we showed above. The order of events in RLogs guarantees the *happens-before* relation between events. The *Happens-before* relation is acyclic. Since in SG, there is an edge from t_i to t_j only if $E_c(t_i) <_s E_c(t_j)$, a cycle in SG will amount to a violation of the *happens-before* relation exhibited in RLogs. Thus, SG is acyclic, hence Message Futures is one-copy serializable. ■

3.3.5 RLog propagation

Message Futures' performance depends on RLogs propagation. The direct influence is that propagation of RLogs controls transactions commit latency. A continuous back-to-back RLog propagation for example will ensure that transactions will commit faster than

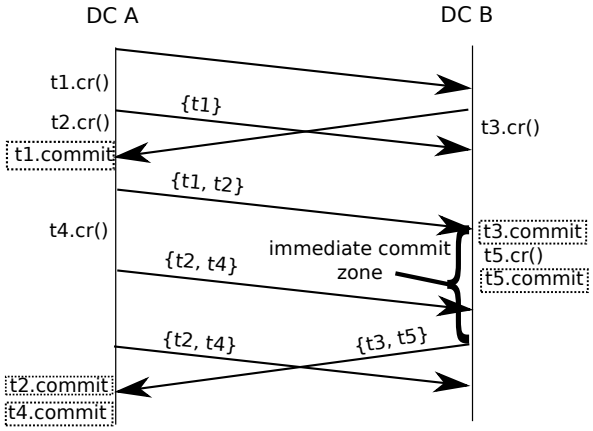


Figure 3.2: Message Futures example scenario of two datacenters with different propagation intervals

a propagation mechanism that delay transmissions. However, it should be noted that scheduling RLog transmissions can enable us to achieve certain desired characteristics. Initially, we will demonstrate this by a simple technique. Consider a propagation controller that sends RLogs at a fixed rate. We will show that assigning different propagation intervals enable us to control the relative performance of different datacenters. Consider the commit condition. We have shown in our first example that immediate commits are possible at a datacenter DC_A when the commit condition is already satisfied when the transaction requests to commit. Lets denote this region where the commit condition is satisfied for a newly committing transaction as the *Immediate Commit Zone* (ICZ). A datacenter, DC_A, is in immediate commit zone if at that point it has already received RLogs from all other datacenters indicating that they know about events up to the current LPT_A. It is clear that longer ICZ durations translates to shorter average commit latency.

We will show with the aid of an example that a datacenter experiences longer ICZ durations compared to datacenters with shorter propagation intervals. The scenario is shown in Figure 3.2, where there are two datacenters, DC_A and DC_B. DC_A propagation interval is one fourth the propagation interval of DC_B. In the figure, the ICZ is labeled. Observe that the cause of this long ICZ duration is that DC_B delays its next RLog

propagation while DC_A, since it is transmitting at a faster rate, has sent an RLog that satisfied the commit condition. The scenario contains five issued transactions. Assume that they are all non-conflicting transactions. Transactions t_1, t_2, t_3, and t_4 all requested to commit in the *regular zone*, opposed to the ICZ. Thus, they all wait for the commit condition to be satisfied. However, note how transactions of DC_B generally commit faster than their counterparts in DC_A. t_5 requests to commit at the ICZ, hence it immediately commits. Increasing the propagation interval of a datacenter to increase its priority harm other datacenters. Careful interval assignments are necessary to achieve the desired properties.

Many factors control how propagation affects Message Futures. Datacenters with shorter propagation intervals suffer less from RLog transmission failures. Also, increasing the propagation interval of even a single datacenter, DC_i, causes the RLog size to increase, since other datacenters are not aware if it has received old events. Understanding the full dynamics and effect of RLog propagation on Message Futures performance is not intuitive and invites a formal analytical treatment. Take note that our discussion here is on propagation mechanisms with a fixed rate. A dynamic propagation protocol will be much more complex, but will definitely allow more flexibility in achieving desired characteristics from the system. This is an undertaken that we are interested in for future work on Message Futures.

3.4 Experimental evaluation

We performed an evaluation of Message Futures on Amazon EC2. The results are compared with a Paxos-based log replication protocol and an optimized version of it, called Paxos-CP [33]. Our implementation of the Paxos-based log replication protocol is based on megastore [37] and we refer to it as the Paxos commit protocol (or simply Paxos) in the

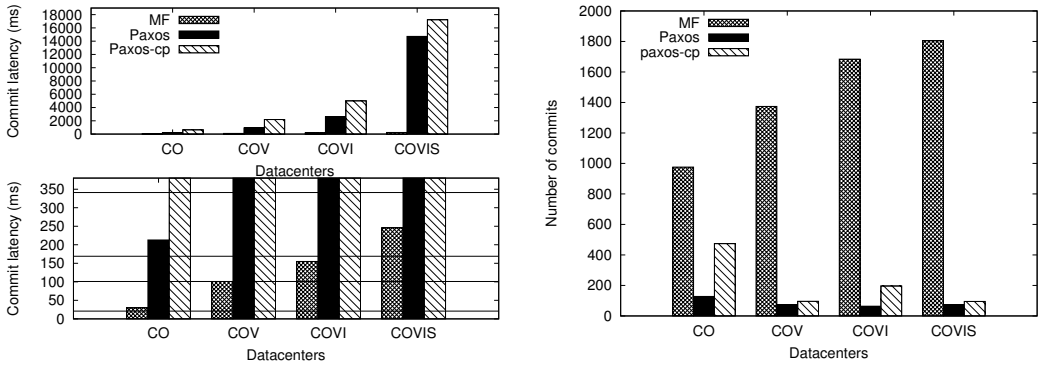
	O	V	I	S
C	21	86	159	173
O	-	101	169	205
V	-	-	99	260
I	-	-	-	341

Table 3.1: RTT latencies between different datacenters in milliseconds.

rest of the chapter. Paxos-CP is a variant of the Paxos commit protocol that allows more concurrency via *promotion* and *combination*. We use machines in five EC2 datacenters: California (C), Oregon (O), Virginia (V), Ireland (I), and Singapore (S). RTT latencies across datacenters are shown in Table 3.1. Each datacenter runs an instance of Message Futures. We run experiments on combinations of those datacenters. A combination is represented by a grouping of the initials of the used datacenters; experiment CI for example is a scenario of two replicas: one in California and the other in Ireland. HBase [88] is used as the underlying key-value store. Yahoo! cloud serving benchmark (YCSB) [89] is leveraged to evaluate the system. Since YCSB does not support transactions, an extended version of YCSB is used to support transactions and transactional workloads [90].

Our experiments focus on measuring the commit latency and the amount of concurrency in the system. Our results show that Message Futures commits transactions, of a datacenter i, quickly with a latency close to RTT^i_{max}, where RTT^i_{max} is the maximum RTT experienced by datacenter i and any other datacenter in the scenario. We use the notion RTT_{max} to denote the maximum RTT^i_{max} for all datacenters i. Other experiments detail the behavior of Message Futures while varying contention, write skewness, and propagation intervals. Also, we report the results of varying propagation intervals in a selected datacenter and observe immediate transaction commits for many cases. Thus, a low average commit latency is demonstrated for the datacenter with the longer propagation interval. Results are detailed next.

Message Futures performance. Our first set of experiments consists of four

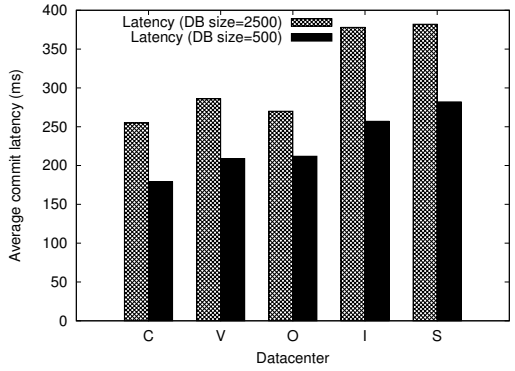


(a) Average commit latency. (b) Number of successful transaction commits.

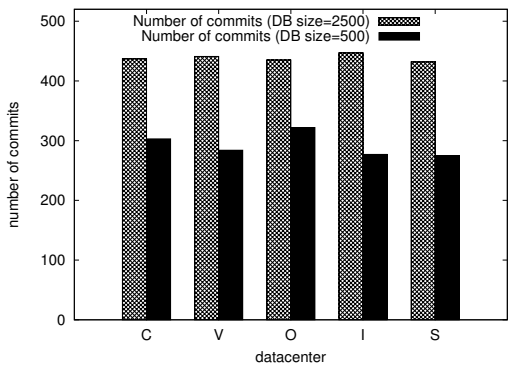
Figure 3.3: Transactions average commit latency and number of commits for scenarios with different numbers of datacenters.

scenarios, CO, COV, $COVI$, and $COVIS$. We will report the average commit latencies and the total number of commits. In this study we compare Message Futures' results to those obtained from Paxos and Paxos-CP. A relatively conservative workload is used for the comparison. This is due to Paxos and Paxos-CP's inability to scale for even such a conservative workload and to enable us to observe the characteristics of Message Futures in the normal case operation. The baseline workload employs five client threads in each datacenter. The rate of requests is 10 transactions per second for each datacenter; for five datacenters the overall rate is 50 transactions per second. We run the experiments for a total of 500 transactions in each datacenter. Each transaction accesses 5 objects uniformly from a pool of 1000 keys. Operations are 50% reads and 50% writes. The interval between two RLog transmissions is called the *propagation interval*. Unless stated otherwise, the propagation interval of any datacenter is set to 100ms, hence 10 RLogs are sent per second.

The results of this experiment are shown in Figure 3.3. Average commit latencies are shown in Figure 3.3(a). The figure is divided into two parts. The top part shows the whole range of obtained results. To enable observing Message Futures results more closely, the lower part is a magnification of the range which shows commit latencies lower than



(a) Latency for committed transactions.



(b) Number of successful transaction commits, out of 500 transactions.

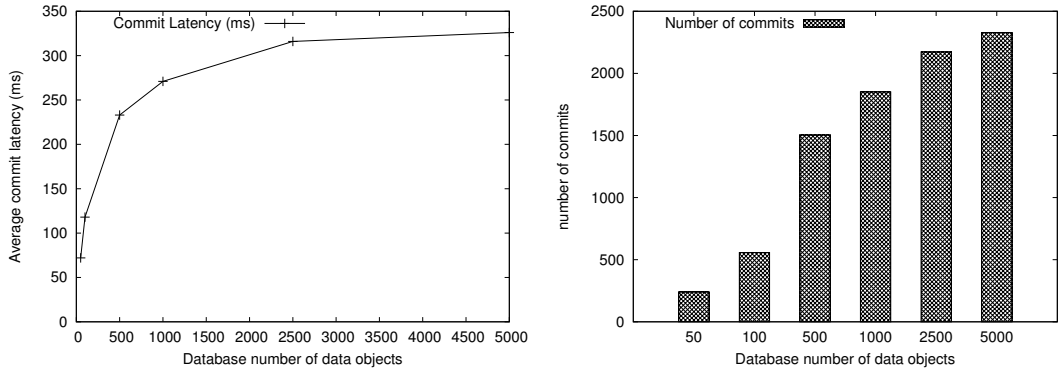
Figure 3.4: Detailed performance of each datacenter in a CVOIS scenario.

360ms. To facilitate comparing with RTT_{max}, RTT_{max} values are plotted as horizontal lines for all scenarios. Commit latencies for Message Futures are close to RTT_{max} and even below it for larger number of datacenters. This is because a datacenter, DC_i, achieves an average commit latency relative to the its RTT^i_{max} value, which is smaller than or equal to RTT_{max}. Also, more aborted transactions yield lower average commit latency, since most aborts in Message Futures are immediate. Message Futures performs considerably better than Paxos and Paxos-CP, where the average commit latency is more than 14 seconds for $COVIS$ compared to 246ms for Message Futures. The number of commits experienced by the system is affected by the increase in the number of datacenters and transactions per second (Figure 3.3(b)). Message Futures, however, manages to achieve an average number of 361 commits in the worst case ($COVIS$), whereas Paxos and Paxos-CP were only able to achieve an average number of 64 and 237 commits respectively in their best case (CO). Note that Message Futures' commit ensures that the commit record is persistent in at least one datacenter, whereas Paxos and Paxos-CP ensures that the commit record is persistent in at least a majority.

The commit latency depends on the issuing datacenter. As we have shown, the commit latency is a function of RTT^i_{max}. However, RTT^i_{max} values are different for datacenters in

the same scenario. From Table 3.1 for example note that RTT^C_{max} equals 173ms, where RTT^S_{max} equals 341ms, almost double the value of datacenter C. Thus, we expect the commit latency of a datacenter to be different than other datacenters according to its RTT^i_{max} value. To illustrate this, results clustered according to the issuing datacenter are shown in Figure 3.4. Two sets of results are shown for different database sizes, namely 500 and 2500 data objects. It is clear that datacenters C, V, and O perform better than their counterparts with higher RTT^i_{max} values. In Figure 3.4(a), the results of individual latencies show that C, V, and O achieve about two thirds the latency of I and S for the case of 2500 data objects. The effect is less apparent for the higher contention case, *i.e.*, where the number of data objects is 500. This is due to the higher abort rate. Since most aborts are either immediate or require much lower latency, they are not as much affected by the wide-area communication latency. A point to consider here is that I and S achieve commit latencies that are close to their respective RTT^i_{max}, whereas C, V, and O are not. This is due the effect of our choice of propagation latency which has a more visible effect for lower RTT^i_{max} values. It will be shown in a later experiment (Figure 3.7) how enforcing a lower propagation interval will make the commit latency of datacenter C close to RTT^C_{max}. The number of commits, shown in Figure 3.4(b), do not show the same variation and dependence on RTT^i_{max}. The number of commits are only slightly better for C and O for scenarios with higher contention.

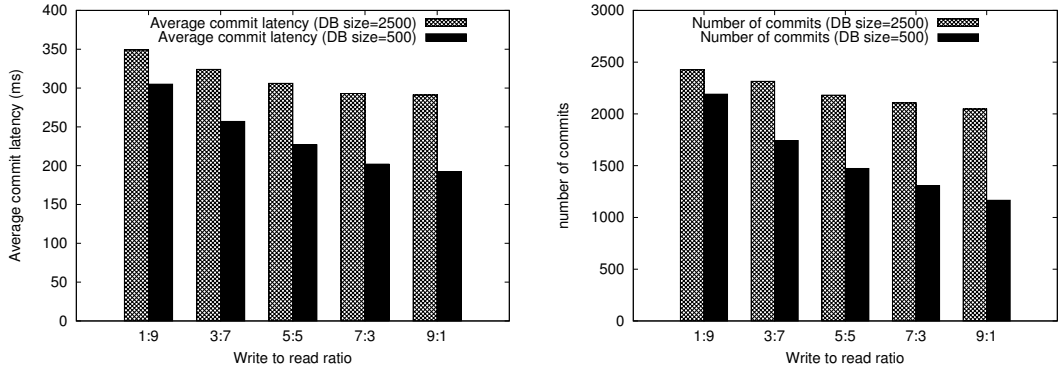
Contention effect. In the next set of experiments we would like to study the effect of contention on the performance of Message Futures. This is tested by varying the size of the database from 50 to 5000 data objects on a $CVOIS$ scenario. Figure 3.5 shows obtained results. Observe how the effect of high contention is apparent for values smaller than 500 data objects. The results then gradually converge to performance results close to scenarios with no contention. For example, the number of commits, as shown in Figure 3.5(b), increase by 7% when the number of data objects is increased from 2500



(a) Latency for committed transactions.

(b) Number of successful transaction commits, out of 2500 transactions.

Figure 3.5: The effect of contention on Message Futures.



(a) Latency for committed transactions.

(b) Number of successful transaction commits, out of 2500 transactions.

Figure 3.6: The effect of Write-to-read ratio on Message Futures.

to 5000. This is at the cost of a 3% increase in commit latency (Figure 3.5(a)). Two main observations are to be taken from these results. First, the effect of increasing the number of data objects diminishes as the size of the database increases. Second, there is a trade-off between commit latency and number of commits with respect to the size of the database. This is an important factor for deciding a suitable granularity for a given database.

Write to read ratio. In all previous experiments an operation can be either a read or a write with equal probability. While this can be considered as a workload with a

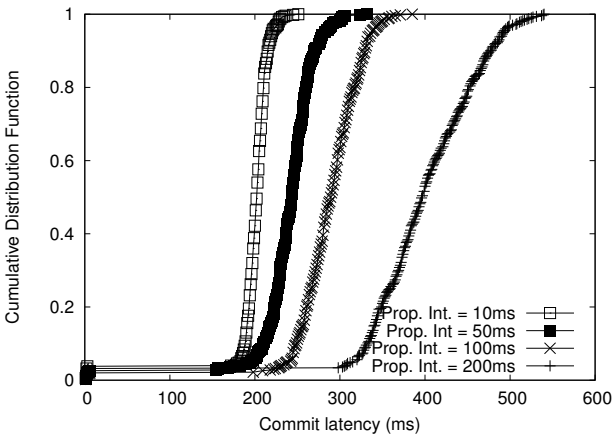


Figure 3.7: Cumulative density function of commit latency of 500 transactions at datacenter C for four global values of propagation intervals.

high number of writes, it is necessary to investigate the behavior of Message Futures for different write to read ratios. The results for varying the write to read ratio are shown in Figure 3.6 for a $CVOIS$ scenario. Two sets of experiments were performed with different database sizes, *i.e.*, 500 and 2500 data objects. Commit latency results, shown in Figure 3.6(a), demonstrate the degrading effect of increasing write to read ratio. This effect is larger for scenarios with higher contention. Consider going from a write to read ratio of 1:9 to 9:1. A degradation of 16.6% is observed for a database with 2500 data objects compared to a degradation of 37% for a database with 500 data objects. The number of commits experiences similar behavior where a degradation of 15.5% is observed for a size of 2500 compared to 46.8% for a size of 500 data objects.

Propagation interval effect. Now we study the effect of the propagation interval on Message Futures. To illustrate this effect, we plot the Cumulative Distribution Function (CDF) of transactions' commit latencies for Datacenter C in a $CVOIS$ scenario for different propagation interval values. In Figure 3.7, the propagation interval is increased from 10ms to 200ms. Note that the propagation interval changed in this set of experiments is for all datacenters, hence it is a global propagation interval. Increasing the propagation

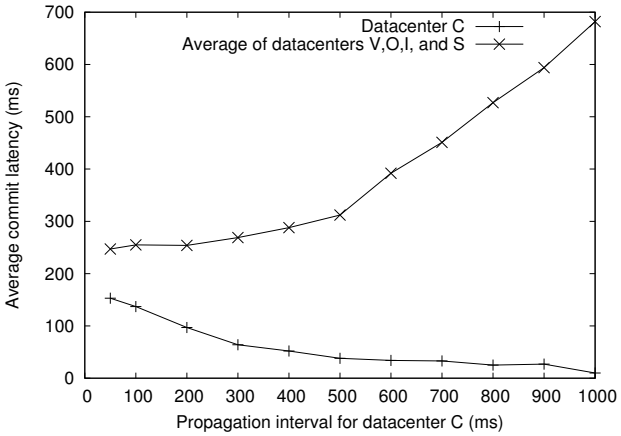


Figure 3.8: Transactions average commit latency while increasing the propagation interval of datacenter C in a CVOIS scenario.

interval causes an increase in commit latencies. Furthermore, increasing the global propagation interval causes the commit latency values to be more variant. For example, with a global propagation interval of 200ms, the bulk of commit latency values range from 300 to 540ms, whereas a global propagation interval of 10ms causes the range of the bulk of commit latency values to be from 170 to 250ms. The range of values of the latter is almost one third of the former.

Performance prioritization In Section 3.3 we showed how a transaction can immediately commit if the commit condition is already satisfied. Commit latency and regions of immediate commits highly depend on the dynamics of transmitting and receiving RLogs. For example, immediate commits will only occur when the commit condition with respect to all other datacenters is satisfied. Smart scheduling of RLog transmission will increase the duration of immediate commits. In the following experiment we use a fixed rate scheduler with different rates to observe the effect of increasing the propagation interval of one datacenter and how that enables us to achieve higher performance results for it. In this experiment, we report the results obtained for C and compare it against the average of all other datacenters in a $COVIS$ scenario. Datacenters other than C have

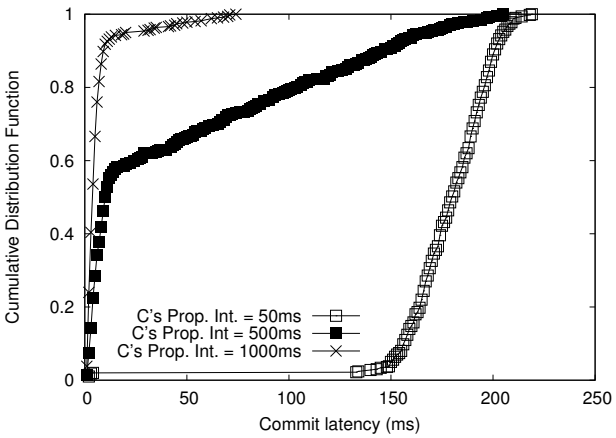


Figure 3.9: Cumulative Distribution Function of commit latencies of 500 transactions in datacenter C with different propagation interval values.

their propagation interval fixed at 10ms in all runs to observe the effect of C's propagation interval in isolation. The results are shown in Figure 3.8. As the propagation interval of C increases, an improvement of C's average latency is experienced. This improvement is at the expense of an increase of the average commit latency of all other datacenters. An important point to observe in C's propagation interval values is 500ms. C's commit latency at this point is 75% less than the run with a propagation interval of 50ms for C. Before this point, the average of other datacenter increases slowly, *i.e.*, an increase of 65ms. However, after the 500ms point, although C's commit latency continue to decrease, this is at a higher cost for other datacenters' commit latencies. Observe that compared to the point at 500ms, when C's propagation interval increases to 1000ms its average commit latency improves by dropping another 74% of its latency to be 10ms whereas the average of other datacenters increase by 119%. Changing the propagation interval does not show a clear effect on the number of commits for C; a number of commits between 376 and 426 is maintained for all runs. The effect on other datacenters is a slowly decreasing average number of commits from 386 to 345 successful commits.

To study the effect of increasing the propagation interval of one datacenter we per-

formed another set of experiments to observe the commit latency values of all transactions. This will allow us to extract the number of transactions that committed immediately and show how other transactions behave. The experiments were performed on a $CVOIS$ scenario where C is the only datacenter issuing transactions. Datacenters other than C transmit RLogs with a fixed interval of 10ms. The results are shown in Figure 3.9. A CDF of obtained commit latency values of scenarios with different propagation intervals are plotted. Note how increasing the propagation interval increases the amount of immediate commits. About 60% and 90% immediate commits are recorded for a propagation interval of 500ms and 1000ms, respectively. Also, note that transactions collectively performs better as the propagation interval increases.

3.5 Concluding Remarks

We proposed Message Futures, a geo-replicated concurrency control manager for multi-datacenter environments. It exhibits low commit latency of transactions, with latency close to the RTT of the system. Proactive message passing of state and transaction information enables datacenters to infer enough information to commit transactions. RLogs are incorporated in the Message Futures design to increase fault tolerance and leverage its conservation of the *happens-before* relation. We demonstrated how our scheme can be used to prioritize the performance of different datacenters. A datacenter with a larger propagation interval experiences lower commit latencies and the immediate commitment of transactions in many cases. An evaluation on an inter-continental setting was performed to demonstrate Message Futures' performance and validate our claims.

Chapter 4

Helios: Achieving Optimal Coordination Latency

4.1 Introduction

This chapter introduces a lower-bound on transaction *commit latency* that is due to the coordination needed to detect conflicts. The commit latency is defined as the time it takes the datacenter to decide whether a transaction executing can be committed or not. A transaction t cannot commit unless the datacenter is certain that there can be no other concurrent transactions that conflict with t, which leads to the need of coordination between datacenters. To ensure the absence of conflicts for two concurrent transactions, at least one of them must know the contents of the other transaction before committing. We use this observation to deduce the lower-bound on commit latency. Basically, a commit latency is *not achievable* if it leads two concurrent transactions to commit without at least one of them learning about the other transaction. As we will demonstrate, if two datacenters have commit latencies that add up to be less than the RTT between them, then it is possible that two transactions commit with both of them being oblivious about

the other. This cannot be allowed because it can lead two conflicting transactions to commit. As we will demonstrate, *for any two datacenters to maintain serializability, the summation of their commit latencies cannot, in any case, be lower than the RTT between them.* The lower-bound result is applicable to a group of datacenters each maintaining a full replica of the data. Any pair of datacenters must satisfy the lower-bound commit latency.

Also, we propose Helios, an optimistic commit protocol influenced by the lower-bound study. It allows manual tuning of commit latency as long as the commit latency does not violate the lower bound. Each transaction is timestamped by a local loosely synchronized clock. The transaction commits by waiting for transaction information from other datacenters. The amount of received information needed to commit a transaction depends on the transaction's timestamp. Datacenters exchange their *preparing transactions* (transactions trying to commit) and *finished transactions* (committed or aborted transactions) using an ordered shared log. As logs arrive, the datacenter decides which local preparing transactions can be committed given the new information. Helios judiciously decides the earliest point in the received logs that will enable committing a transaction. Recognizing the *earliest* point in the received logs to commit a transaction will lead to lower commit latency. We focus in this chapter on achieving the lowest *average* commit latency. However, Helios allows tuning the commit latencies of individual datacenters to achieve other objectives.

Helios cleanly separates the protocol to guarantee serializability from the mechanism to ensure liveness in the presence of failures. This design is followed by other geo-replicated protocols such as Spanner [25] and Scatter [53] that use Two-Phase Commit (2PC) to guarantee consistency and Paxos [26, 27] to perform state replication. Helios coordinates between all datacenters to commit transactions. If a datacenter outage occurs, the commit protocol will block preparing transactions. This is similar to blocking scenarios of 2PC.

This blocking is due to the non-failed datacenters waiting for information from the failed datacenters. To overcome failures, Helios leverages a separate synchronous replication component and augments it with the Helios commit protocol. This clear separation allows the flexible use of any replication protocol such as Paxos. Paxos, however, requires two rounds of communication. Also, it replicates to a majority of datacenters and does not provide any flexibility in setting the number of tolerated datacenter outages. We design a state replication protocol that utilizes the replicated log used for the commit protocol to ensure liveness in the presence of failures while allowing the flexibility of setting the number of tolerated datacenter outages.

Helios does not rely on clock synchronization for its correctness. However, as we will show in the chapter, better clock synchronization will result in better commit latency of transactions. Traditionally, the solution space for practical geo-replicated data stores did not include the use of synchronized real-time clocks. This design principle is now under scrutiny. It has been shown in Spanner [25], Google's globally-distributed database, that achieving real-time synchronization with low error margins is possible for data management purposes. Spanner's synchronization manager, TrueTime, uses GPS, atomic clocks and relies on a variant of NTP [91] to achieve synchronization. In our evaluation, we use a readily available NTP client to perform synchronization with no additional hardware and observe that this level of synchronization allows us to achieve the desired performance. However, integrating advanced clock synchronization hardware will improve accuracy.

Section 4.2 derives a lower bound on transactions running on replicated data. The Helios commit protocol and replication are described in Section 4.3 followed by discussions about the performance of Helios in Section 4.4. Then, evaluation results are presented in Section 4.5. The chapter concludes in Section 4.6.

4.2 Commit latency lower-bound

The objective of this section is to develop a lower-bound on commit latency of transactions on replicated data stores *while maintaining serializability* [12]. Maintaining serializability requires coordination between replicas (datacenters in our case). The communication latency necessary for this coordination imposes a limit on *commit latency*, which is the time duration to decide whether a transaction commits or aborts. Achieving low commit latency is the focus of this study. Consider two datacenters A and B with unique commit latencies L_A and L_B, respectively. We show in this section that the *summation* of L_A and L_B must be at least the Round-Trip Time (RTT) between A and B. Note that this is a summation which means that the commit latency of a datacenter can be lower than RTT.

The lower-bound result extends to larger groups of datacenters by applying the lower-bound to all pairs in the group. This will allow us to judge whether the group of datacenters can commit with a certain set of commit latency values. We are particularly interested in minimizing the average commit latency of all datacenters. We call the minimum average latency a *Minimum Average Optimal* (MAO) latency or optimal latency for short.

4.2.1 Theoretical model and assumptions

We consider a theoretical model that consists of datacenters with communication links connecting them. Each transaction undergoes two phases. First, the transaction is issued and it becomes visible to the datacenter. At that stage it is called a *preparing* transaction. Then, at a later time the datacenter decides whether it commits or aborts and it becomes a *finished* transaction. The time spent as a preparing transaction is the *commit latency*.

The following are the assumptions on communication and computation for this model.

These assumptions are made for the theoretical development of this section only and are not part of the Helios system design in Section 4.3.

- *Compute power:* Infinite compute power is assumed in the model. The datacenter does not experience any overhead in processing and storing transactions. We make this assumption to focus our attention on communication overhead.

- *Communication links:* Sending a message through a link takes a specific latency to be delivered to the other end. Links are symmetric and take the same amount of time in both directions. Note that different links could have different latencies. However, triangle inequality must hold.

- *Arbitrary read-write transactions:* All datacenters have no restrictions on their choice or order of objects to be read or written in a transaction. Additionally, each transaction must have at least a single write operation. Thus, the model does not apply to optimizations for read-only transactions and disjoint data manipulation techniques. Also, transactions must try to commit, hence aborting all transactions is not allowed.

- *Knowledge:* Each datacenter A knows precisely every preparing and finished transaction that exists at another datacenter B up to the current time minus half the RTT between them, *i.e.*, *now* - $\frac{RTT(A,B)}{2}$. This reflects the fastest time a datacenter knows about any event in another datacenter. In a realistic setting this is a lower bound of such knowledge.

- *Commit latency:* We assume that the commit latency at each datacenter is fixed. This assumption simplifies the presentation. The discussions can be extended to the general case by taking each point in time in isolation.

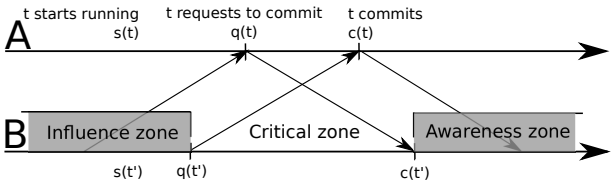


Figure 4.1: Two transactions, t and t', executing in a scenario with two datacenters

4.2.2 Lower-bound proof

Intuition. For any two concurrent conflicting transactions, at least one of them must be able to detect the other before committing. Otherwise, both transactions will commit, which could result in incorrect executions. Here, we show that there is a lower-bound on commit latency. If the commit latency is lower than the lower-bound, then two conflicting transactions could commit without detecting each other, thus possibly violating correctness.

Formulation. Consider two datacenters A and B and a transaction t executing at datacenter A and transaction t' executing at datacenter B that could be conflicting with t. Figure 4.1 shows these transactions. In the figure, $s(t)$ is the transaction's start time, $q(t)$ is the commit request time, and $c(t)$ is the commit time. Transaction t's read and write-set are visible at the commit request time. Given the knowledge assumption, B knows about t starting from time $q(t) + \frac{RTT(A/B)}{2}$. Transaction t is preparing from time $q(t)$ until time $c(t)$ when it is committed. Three *zones* are defined at datacenter B with respect to t: (1) The *awareness zone* where B can possibly know about t, (2) The *influence zone* where B's transactions can be known to t, and (3) the *critical zone* where B is neither in the awareness nor influence zone.

Lemma 1 *The sum of the commit latencies of two datacenters is greater than or equal to the RTT between them,* i.e., $L_A + L_B \geq RTT(A, B)$*, where* L_X *is the commit latency*

at datacenter X.

Proof: Let the time when t requests to commit be $q(t)$ and the time it commits be $c(t)$, *i.e.*, $c(t) - q(t) = L(t)$. These times are illustrated in Figure 4.1. The earliest time that B can be aware of the commit request made by t is at time $q(t) + \frac{RTT(A,B)}{2}$, since half the RTT is needed to get a message from A to B. Thus, t can affect B only at the *awareness zone* which is at any time greater than or equal to $q(t) + \frac{RTT(A,B)}{2}$. Likewise, t cannot be affected by events at B that happened after time $c(t) - \frac{RTT(A,B)}{2}$ since they cannot be received at A before the commit decision is made. We denote the times when an event at B can affect the outcome of t as the *influence zone* which is any time less than or equal to $c(t) - \frac{RTT(A,B)}{2}$.

Now consider the time duration that is neither in the awareness zone nor in the influence zone. Call this time duration the *critical zone*. Consider a transaction t' at B that requests to commit and commits in the critical zone. Transaction t' will not affect the outcome of t, since t' is not in the influence zone. Also, t will not affect t', since t' is not in the awareness zone. Assume that t' can successfully commit. However, t' can conflict with t. Since t' is not aware of t and t is, likewise, not aware of t', both transactions successfully commit. This potentially results in an inconsistency, a contradiction to the assumption that t' can successfully commit. This means that a transaction that starts at the beginning of the critical zone at B cannot commit with a commit latency smaller than the duration of the critical zone. This duration is equal to $RTT(A, B) - L(t)$.

Repeating the same steps above for each point in time at datacenter A will yield that the commit latency at A, L_A, is equal to $L(T)$ and the commit latency at any point in B, L_B, is larger than or equal to $RTT(A, B) - L_A$. Thus, the sum of L_A and L_B must be greater than or equal to $RTT(A, B)$. ■

The previous lemma shows that there is a direct trade-off between the commit latencies

Protocol	L_A	L_B	L_C	Average
Master/Slave (A master)	0	30	20	16.67
Master/Slave (C master)	20	40	0	20
Majority	20	30	20	23.33
Optimal (MAO)	5	25	15	15

Table 4.1: Possible commit latencies, L_A, L_B and L_C, for three datacenters with Round-Trip Times $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$.

of two datacenters. Given this lemma we are now able to judge whether a set of commit latencies are *achievable* or violates the lower-bound for scenarios with more than two datacenters by applying the lower-bound to each pair of datacenters.

Example. Consider an example of three datacenters, A, B, and C. The RTTs between the datacenters are: $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$. Table 4.1 shows four achievable commit latencies and the average commit latency of the datacenters. The first two represent a master-slave replication approach, where a single master is responsible for committing transactions. In this approach, the master commits immediately, and the other datacenters commit latencies are the RTT to the master. Note how each pair of datacenters satisfies the lower-bound, *e.g.*, when A is the master $L_A + L_B = 30 = RTT(A, B)$. The third row represents a majority replication approach. For the case of three datacenters, the commit latency of a datacenter is the RTT to the nearest datacenter. These replication protocols experience different average commit latencies: 16.67, 20, and 23.33. However, the minimum average commit latency (MAO) that is achievable for this scenario is 15. The fourth row in the figure show the commit latencies, L_A, L_B, and L_C, that achieve an average commit latency of 15 while not violating the lower-bound.

Deriving the achievable minimum average commit latency for a given set of datacenters is outlined next.

4.2.3 Minimum average optimality

A MAO set of commit latency values minimizes the average commit latency of all datacenters without violating the lower-bound condition (Lemma 1) for any pair of datacenters. The MAO solution can be derived using the following linear programming formulation:

Problem 1 ***(Minimum Average Optimal)***

The Minimum Average Optimal commit latencies for n datacenters is derived using a linear program with the following objective and constraints:

$$\begin{array}{lll} \text{Minimize} & \sum_{A \in R} & L_A \\ \text{subject to} & \forall_{A,B \in R} & L_A + L_B \geq RTT(A,B) \\ \text{and} & \forall_{A \in R} & L_A \geq 0 \end{array}$$

where R is the set of datacenters. This formulation follows directly from Lemma 1. Minimizing the latency is our objective and the constraints are the correctness conditions that commit latencies are not negative and Lemma 1 is satisfied. We will use this methodology to derive the commit latency values used with the Helios commit protocol. This linear program can be adapted to other objectives. In Section 4.4.2, we discuss optimizing for throughput.

4.3 Helios commit protocol

In this section, we propose Helios [8]. The protocol design and operation are developed followed by a discussion on handling datacenter outages.

4.3.1 Helios architecture

System model. We consider a multi-datacenter system consisting of datacenters and clients. Each datacenter contains a *full copy of the data* and runs an instance of Helios, which is an optimistic concurrency control manager. Read operations are performed by clients first, and write operations are buffered. When the client is ready to commit, it sends a commit request containing its read and write-sets to the closest Helios instance. Helios replies to read requests with the current version and version timestamp of the requested data object. The version timestamp is the timestamp of the most recent write operation that wrote the data object. Blind writes are allowed, meaning that an object can exist in the write-set without being read. The commit request contains the read-set with the read version timestamps and the buffered write-set. Helios upon receiving the commit request will start the commit protocol to commit the transaction and propagate all updates to other datacenters. After committing the transaction, the commit decision is sent back to the client. *The time spent by the client from sending the commit request to receiving the commit decision is called the* ***commit latency***.

Communication. Helios uses a *log replication protocol* to exchange transaction information between datacenters. The log is continuously being propagated between datacenters. Each record in the log contains the information of either a *preparing transaction* that is trying to commit or a *finished transaction* that is either committed or aborted. Each transaction has two records in the log, one added when it starts as a preparing transaction and one record when it becomes a finished transaction. The transaction information includes the read and write sets. Each transaction is timestamped. The log is ordered according to transaction timestamps. Furthermore, *records are received by other datacenters according to their order in the log.* This means that receiving a record of a transaction with timestamp τ will follow all transactions with lower timestamps.

Timestamps reflect the local clock of the datacenter. Clocks are loosely synchronized.

Helios conducts this replication using a replication protocol that is similar to Replicated Dictionary (RDict) [87][1]. RDict is an efficient protocol to replicate logs while maintaining their order. A $N \times N$ timetable, T_A, is maintained at each datacenter A where N is the number of datacenters. Each entry in the timetable is a timestamp representing a bound on how much a datacenter knows about another datacenter's records. For example, entry $\mathbf{T_A}[\mathbf{B}, \mathbf{C}] = \tau$ means that datacenter A knows that datacenter B is aware of all events at datacenter C up to timestamp τ. This notation will be used while describing Helios.

4.3.2 Helios overview

Intuition

To provide an intuition of the Helios commit protocol, consider the scenario in Figure 4.1. The figure shows the timeline of two datacenters, A and B. At A, a transaction t is issued at time $q(t)$ and committed at time $c(t)$. Transaction t commits immediately after receiving a log from B that is shown as an arrow going from B to A. This log carries transactions that were issued up to the time of sending the log, including transaction t' (assume t' is issued at the time of log transmission). The time the log was sent from B is $q(t')$. $q(t')$ is also the commit request time of t'. Helios receives the log in order, meaning that all transactions, preparing or finished, at B prior to or at time $q(t')$ are known to A at time $c(t)$.

Detecting conflicts. Transactions at B must not conflict with t. The approach to avoid conflicts is influenced by the way the lower-bound latency was developed in Section 4.2. However, here we do not make any assumptions regarding clock synchronization or communication. Rather, we rely on the exchanged logs and received transaction

[1]The Replicated Dictionary was introduced as the Replicated Log in Section 3.2.3.

timestamps.

A transaction, t', at B is either issued during the influence zone, critical zone, or awareness zone. If t' starts during the influence zone, then transaction t will detect it because the log will contain a record of t'. If t' starts in the awareness zone, then it will detect t. Thus, for these two cases, conflicts will be detected. An undetected conflict can arise only if t' starts *and* commits within the critical zone. Thus, if t' is issued in the critical zone, Helios must ensure that it does not commit until it is in the awareness zone, which means that B will detect the conflict between t and t'.

Commit offsets. A *commit offset* is a time duration that represents the extent of knowledge needed by a transaction from other datacenters prior to committing. This duration ensures that a transaction t' that is issued in the critical zone will commit in the awareness zone (more on how to optimally assign these commit offsets in Section 4.3.5). Each datacenter maintains a commit offset for all "other" datacenters. For example, in the figure, A has a commit offset for B, denoted co_A^B, and B has a commit offset for A, denoted co_B^A.

Committing. When a transaction, t, requests to commit, it is assigned a timestamp, $q(t)$. Helios uses $q(t)$ and co_A^B to calculate a timestamp called the *knowledge timestamp* (kts). There is a kts value for every other datacenter B:

$$kts_t^B = q(t) + co_A^B \tag{4.1}$$

The commit condition for t is the following: *t can commit if its datacenter, A, knows about the transactions at every other datacenter B that were issued up to time kts_t^B at B.*

Consider Figure 4.1. When t is issued, Helios records that $kts_t^B = q(t) + co_A^B$. Assume that kts_t^B is equal to $q(t')$. Thus, A waits until it receives the log sent at $q(t')$ to commit. In the figure, t receives that log at time $c(t)$ and is then able to commit.

Condition on commit offset

In order to guarantee that a transaction t' that is issued in the critical zone will commit in the awareness zone, Helios needs to enforce a condition on the assignment of commit offsets.

Consider the scenario in Figure 4.1. Assume that the commit request time of t', $q(t')$, is in the beginning of the critical zone, as is shown in the figure. Transaction t', similar to t, is assigned a knowledge timestamp when it requests to commit. This value for t' is: $kts^A_{t'} = q(t') + co^A_B$. Thus, t' will know about all transactions at A up to time $kts^A_{t'}$ at A. For t' to be aware of t before committing, the value $kts^A_{t'}$ must be greater than or equal to the time t was issued, *i.e.*, $q(t)$. By expanding the value of $kts^A_{t'}$, the inequality to guarantee detecting conflicts is: $q(t')+co^A_B \geq q(t)$. $q(t)$ is equal to $q(t')-co^B_A$. Substituting this into the inequality and rearranging, the inequality becomes: $(co^B_A + co^A_B) \geq 0$. This is summarized by the following rule:

Rule 1 *For the Helios commit protocol to be able to detect conflicts, the sum of any two symmetric commit offsets (*e.g.*, $co^B_A + co^A_B$) must be greater than or equal to 0.*

The requirement above specifies what is necessary to ensure correctness. However, a range of possible commit offset assignments might be used. We show later in Section 4.3.5 how Helios assigns commit offsets to minimize commit latencies. This assignment by Helios can theoretically achieve the lower-bound commit latency. Note that although timestamps are used, time synchronization is *not* required for correctness. Nonetheless, better synchronization will yield better performance as we demonstrate in Section 4.4.

Example scenarios

To better illustrate how Helios works, consider the example in Figure 4.2 with two datacenters A and B. Time is denoted by the number between A and B's timelines. A

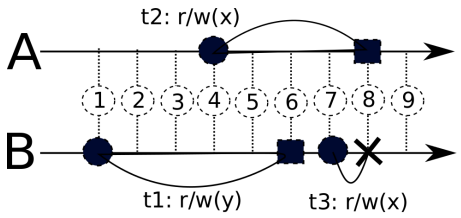


Figure 4.2: A scenario of Helios. Commit latencies are 3 and 5 for A and B. RTT is 8. A circle is a commit request, the square is a commit, and an X sign is an abort.

commit request is denoted by a black circle and is connected to commit or abort time represented as a black square for a commit or an X sign for an abort. Transaction information is displayed as the read and write-sets. For example, transaction t_1 reads and writes y. Assume that the read version is the latest available version prior to requesting the commit. Log transmissions are omitted from the timeline, but assume for this example that they are being propagated at each tick of the clock. The RTT between A and B, $RTT(A, B)$, is equal to 8 time units. To simplify the presentation of the example, assume that the log takes exactly 4 time units to be delivered. Thus, at time 5, B knows about all events at A up to time 1.

For this example we pick the commit offset values to be -1 for co_A^B and +1 for co_B^A, hence $co_A^B + co_B^A$ is greater than or equal to 0. Now, follow the example scenario. Transaction t_1 reads and writes y and requests to commit at time 1, hence $q(t_1) = 1$. The knowledge timestamp for t_1 is given by: $kts_{t_1}^A = q(t_1) + co_B^A$ which is equal to 2. The transaction waits until B receives the log sent from A at time 2, which is the value for $kts_{t_1}^A$. At time 6, t_1 successfully commits after receiving the log from A that was sent at time 2 (the log takes 4 time units to be received). t_2, which reads and writes x, requests to commit at A at time 4. The knowledge timestamp for t_2 is given by: $kts_{t_2}^B = q(t_2) + co_A^B$ which is equal to 3. Assume that the log transmission from B at time 3 took more time than usual and was received at A at time 8, one time unit late. For this reason, t_2 commits at time 8 when the history of datacenter B is received up to time 3, which is the value of $kts_{t_2}^B$.

Algorithm 3: Processing commit requests at A

```
1: t := local transaction requesting to commit at A
2: if t conflicts with any t' ∈ PTPool ∪ EPTPool then
3:     Abort t; exit
4: for each object o in t.readset do
5:     if o is overwritten then
6:         abort t; exit
7: t.timestamp = get_time()
8: for each datacenter X ∈ datacenters do
9:     t.kts^X = t.timestamp + co_A^X
10: PTPool.append(t); Log_A.append(t)
```

Now consider t_3. It requests to commit at B at time 7. The knowledge timestamp value is given by: $kts^A_{t_3} = q(t_3) + co^A_B$, which is equal to 8. However, one time unit after its request, it receives t_2's information that was sent at time 4 from A. A conflict is detected and t_3 aborts immediately.

4.3.3 Concurrency control protocol

In this section, we discuss the design of Helios. The main tasks performed are: (1) process commit requests (Algorithm 3), (2) process remote transactions received in the shared log (Algorithm 4), and (3) commit preparing transactions (Algorithm 5). We also briefly discuss read-only transactions in Section 4.4.3.

Commit requests

When Helios receives a commit request for a transaction t at datacenter A, it checks whether it conflicts with preparing transactions (Lines 2-3). Preparing transactions are maintained in the *Preparing Transactions Pool* (PTPool) for local transactions and the *External Preparing Transactions Pool* (EPTPool) for remote transactions. Transaction t aborts if a conflict exists, which is an intersection between the read or write-set of t with the write-set of any preparing transaction. Then, the read-set of t is verified to have not been

Algorithm 4: Processing transactions in the log at A

```
1:  for each transaction t in Log_A do
2:      if t is local then
3:          skip to next
4:      if t conflicts with any t' ∈ PTPool then
5:          abort t'; t'.timestamp = get_time()
6:          Log_A.append(t')
7:      if t.type == preparing then
8:          EPTPool.append(t)
9:      else // t.type == finished
10:         if t.committed==true then
11:             for each object o in t.writeset do
12:                 Apply o to data store
13:         Remove t from EPTPool
14:     T_A[A, host(t)] = t.timestamp
```

overwritten (Lines 4-6). If no conflicts are detected and the read-set is not overwritten, the knowledge timestamps are calculated as defined in Equation 4.1 (Lines 7-9). A preparing record of t is appended to both the PTPool and the local log, Log_A (Line 10). Appending to the log includes adding the record and updating the timetable so that $T_A[A, A]$ equals to t's timestamp.

Log processing

Helios processes transactions in the log in order (Algorithm 4). Local transaction records are not processed (Lines 2-3). For a coming transaction t, conflicts are detected with local preparing transactions in PTPool. A conflict exists if the read or write set of a local preparing transaction, t', intersects with the write-set of t (Lines 4-6). A conflicting transaction in PTPool is aborted by changing its state to aborted and updating its timestamp. An abort record is added to the log (Lines 5-6). Remember that adding to the log includes updating the timetable to reflect the addition of a new transaction record.

Remote transactions are either preparing or finished. When a preparing transaction is received from another datacenter, it is added to EPTPool (Lines 7-8). A finished

Algorithm 5: Committing preparing transactions at A

```
1: for each transaction t in PTPool do
2:     for each datacenter X ∈ datacenters do
3:         if T_A[A, X] < t.kts^X then
4:             skip to next transaction
5:     Apply t.write-set to local data store at A
6:     commit t; t.timestamp = get_time();
7:     Log_A.append(t)
```

transaction contains a flag to indicate whether it has committed or aborted. If it is committed, then the write operations in the write-set are applied to the local data store (Lines 11-12). However, whether a finished transaction is committed or aborted, it is removed from the EPTPool (Line 13). Finally, the timetable is updated to reflect that t is processed (Line 14).

Committing preparing transactions.

A preparing transaction, t, can successfully commit by satisfying two conditions: (1) *External knowledge:* Helios must have processed transactions from other datacenters up to the *knowledge timestamp* (kts) calculated according to Equation 4.1. (2) *Conflict freedom:* no conflicts were observed with t up to the point when the first condition is satisfied. Algorithm 5 checks whether the external knowledge condition is satisfied for transactions in PTPool (Lines 2-4). If the condition is satisfied for a transaction t, then it can successfully commit. The write-set of t is applied and a record is added to the log (Lines 5-7). Conflicts are already detected when the transaction requested to commit (Algorithm 3) and while the log is being processed (Algorithm 4). Thus, there is no need to detect conflicts at this point.

Commit condition. The commit condition can be summarized as the following:

Rule 2 *A transaction t in A commits if no conflicts are detected, and*

$$T_A[A, B] \geq kts_t^B, \quad \forall_B (B \in R)$$

where R is the set of other datacenters and kts is the knowledge timestamp defined in Equation 4.1.

4.3.4 Liveness

Intuition. Helios needs information from other datacenters to be able to commit its preparing transactions (see Rule 2). An outage of a datacenter will cause other datacenters to block waiting for its transaction log. The blocking will continue until the datacenter is back up again and Helios is recovered. This is similar to blocking scenarios in 2PC. State machine replication (SMR) is used to overcome these blocking scenarios. Here, we present the way Helios achieves liveness while enabling the flexibility to set the number of tolerated datacenter outages. Thus, Helios enables controlling the trade-off between liveness and performance.

The main idea is to ensure that a transaction, t, at datacenter A does not commit before its information exists at f other datacenters, where f is the number of datacenter outages to be tolerated. Verifying that t's information exists at another datacenter, B, can be done by waiting for an acknowledgment of the receipt of t. B should keep the information about t until it is completed and its information are propagated to other datacenters. This is important to enable B to propagate the information about t in case A experiences an outage.

Committing. A datacenter can know whether another datacenter received the record of a transaction by examining the transaction's timestamp and comparing it to the extent of the other datacenter's knowledge using RDict (see Section 4.3.1). If the number of datacenter outages that are to be tolerated is f out of n total datacenters, then the transaction waits for its information to be received by f other datacenters.

Failure case. When a datacenter B fails, a datacenter A might have to delay the

commitment of a transaction until it can ascertain the identity of all previously finished transactions at the failed datacenter B by communicating with other datacenters, C, where backup information for B exists. However, it is not always possible to conclude that a datacenter has failed since a network partition might render the messages sent from B undelivered. Consider the following scenario: a datacenter A is waiting for information from B. A network partition makes information from B unable to be delivered to other datacenters. Given that no information is received at A from B, datacenter A consults C for information about B's finished transactions. Datacenter A can commit transactions since it knows that B cannot commit any transactions without getting an acknowledgment of its receipt from either B or C.

Grace time. The subtlety here is about knowing when a datacenter C can be certain about the state of another datacenter (B). Helios adopts a *time-based invalidation* technique to enable a datacenter to ascertain the state of a failed datacenter. Helios makes the commitment of a transaction dependent on the reception time at the other datacenters. We call this time the *Grace Time* (GT). A datacenter C will acknowledge the receipt of transaction t from B if and only if the transaction information is received at C at time τ that is smaller than the commit request timestamp of t plus GT, *i.e.*, $\tau \leq (q(t) + GT)$. Otherwise, C will not acknowledge the reception of t.

The implication of this bounded acknowledgment is that at time τ at datacenter C, we have a guarantee that no transactions with a request timestamp less than (τ-GT) will be able to synchronously replicate to C. Thus, B and C will be able to infer information about A. Specifically, B and C will be confident that no transactions (unknown to them) will commit at A with a timestamp less than $min\{now_B, now_C\} - GT$, where now_X is the current time at datacenter X. This is because even if a transaction with an earlier timestamp existed at A it will be invalidated since it was not acknowledged by B and C. This is very useful information for Helios because it allows transactions at B and C to

commit even if A fails or cannot communicate with other datacenters.

Integration. This acknowledgment mechanism can be easily incorporated in Helios by extending commitment Rule 2 to reflect the acknowledgment requirement and invalidation. First, the extent of A's knowledge of B's events, $T_A[A, B]$, can be inferred by other participants. Thus, when $T_A[A, B]$ is used in Rule 2 it can be substituted by:

$$\hat{T}_A[A, B] = max\{T_A[A, B], \eta\} \tag{4.2}$$

where η is the extent of knowledge of A about B that can be inferred using a set of any $n - f$ other datacenters called κ. η is calculated as the following:

$$\eta = min\{T_A[C, C]\}\forall_{C\in\kappa} - GT \tag{4.3}$$

The other needed extension to Rule 2 is to restrict the commitment of a transaction until it has been successfully acknowledged by a number of datacenters equal to the number of outages to be overcome. A transaction t at A is considered acknowledged by B if $T_A[B, A]$ is greater than or equal to t's timestamp: $q(t)$. Thus, what is needed is to ensure that the transaction was acknowledged by a set of $f + 1$ total datacenters. However, remember that for a transaction to be successfully acknowledged it needs to be received by the other datacenter in a bounded time, hence $q(t) + GT$. This is ensured by examining $ts_C(t)$, which is the time the transaction record of t was received at C. Given the aforementioned extensions, the commitment rule can now be written as the following:

Rule 3 *A pending transaction t in A commits if no conflicts are detected, and*

(1) $\hat{T}_A[A, B] \geq kts_t^B, \quad \forall_B(B \in R)$

(2) $T_A[C, A] \geq q(t), \quad \forall_C(\ C \in \kappa')$

(3) $ts_C(t) < q(t) + GT, \quad \forall_C(\ C \in \kappa')$

where κ' is a set of any $n - f$ datacenters where n is the number of datacenters and f is the number of tolerated outages.

Grace time. The choice of the value of GT is controlled by the trade-off between minimizing the effect of datacenter outages on progress and minimizing the number of aborted transactions due to delays in the acknowledgment process. A datacenter outage increases the latency because rather than waiting for the knowledge timestamp (Equation 4.1) from the failed datacenter, a datacenter has to wait for an additional duration of GT, to accumulate knowledge of the transactions in the failed datacenter from all the running ones. Also, having a small GT might lead to some transactions unnecessarily aborting due to message delays or drops while waiting for the acknowledgment.

4.3.5 Commit offsets assignment

The assignment of commit offsets directly influences the commit latency of each datacenter. It is not straightforward to assign commit offsets to minimize commit latencies. However, it is possible to estimate commit latencies given the assigned commit offsets. To simplify this estimation we assume that clocks are synchronized and that messages take exactly $\frac{RTT(A,B)}{2}$ time to be sent from A to B. This is of course not true of real systems and will introduce an estimation error of the commit latency.

Estimating commit latency. A transaction t at A waits for information from other datacenters. For every other datacenter B, t can commit only after receiving information from B up to time $(q(t) + co_A^B)$. This information is received at A at time $(q(t) + co_A^B + \frac{RTT(A,B)}{2})$. Since, t commits after receiving information from all datacenters, the commit latency of t is delayed until the time it receives the information from the last datacenter; if that datacenter is C, then the commit latency of transactions at datacenter

A is

$$L_A = co_A^C + \frac{RTT(A,C)}{2} \tag{4.4}$$

Setting commit offsets. Helios can assign arbitrary values to commit offsets, thus targeting the desired commit latency as long as they do not violate the commit offset correctness requirement (Rule 1). The linear program in Section 4.2.3 can be used to derive the commit latency values that will minimize the commit latency. Thus, we can set the commit offsets by the following rearrangement of Equation 4.4

$$co_A^C = L_A - \frac{RTT(A,C)}{2} \tag{4.5}$$

where L_A is the target commit latency calculated using the linear program in Section 4.2.3 and $RTT(A,C)$ is an estimation of the RTT.

Correctness. The optimal assignment is guaranteed to satisfy the correctness requirement of commit offsets shown in Rule 1. This can be verified by substituting Equation 4.5 into the sum requirement (Rule 1) that $(co_A^C + co_C^A)$ must be greater than or equal to 0. This will yield to the requirement that ($L_A + L_C$ - $RTT(A,C)$) must be greater than 0. Since the linear program of Section 4.2.3 has a constraint that the sum of two commit latencies must be greater than the RTT between them, this yields that ($L_A + L_C$ - $RTT(A,C)$) is always greater than 0. Thus, the assignment guarantees detection of conflicts.

Estimation vs. reality. In a real deployment the assignment in equation 4.5 will not yield the exact lower-bound commit latency because of communication links variability and lack of perfect synchronization. We quantify some of these effects in Section 4.4 and show in the evaluation (Section 4.5) that a real-life deployment is able to achieve a commit latency close to the optimal.

4.4 Discussions About Helios' Performance

In this section, we provide an analysis of the factors affecting the commit latency and the trade-off between commit latency on one hand and liveness or throughput on the other hand in addition to optimizing read-only transactions.

4.4.1 Commit latency analysis

This analysis shows how loose synchronization, communication links and compute variability affect the observable commit latency of Helios. We assume that Helios is assigned the *correct* commit offsets, which would lead to the exact lower-bound commit latency in perfect conditions.

Synchronization. Helios does not require clock synchronization for its correctness. However, the degree of synchronization affects the achieved commit latency. In particular, a low level of synchronization or the lack of synchronization will lead to degraded performance. Better clock synchronization will lead to commit latency values that are closer to the optimal. Recall that the commit condition in Rule 2 states that a transaction must wait for every datacenter to send a message with a timestamp calculated by using the commit offsets. Using this protocol will result in the transaction committing with the commit latency estimated in Equation 4.4. However, with a clock skew between datacenters, the observed commit latency will be affected. Clock skew between two datacenters, A and B, will cause one of them to receive the expected message later than intended and the other will receive it earlier than intended. Thus, the actual time that a datacenter A waits until it receives the expected message from B for a transaction t is $L_A + \theta(A, B)$, where $\theta(A, B)$ is the time difference between A and B. $\theta(A, B)$ has a positive value if A's clock is ahead of B's clock. It is possible that A receives the message prior to requesting the commit; this is the case when $\theta < -L_A$. With the presence of

clock skew the actual observed commit latency at A, $\bar{L_A}$, is given by

$$\bar{L_A} = L_A + Max_{B \in R}\{\theta(A, B)\} \tag{4.6}$$

RTT estimate accuracy. RTT estimation also plays a role in determining the achieved commit latency. To analyze the effect of RTT estimation errors, assume that the difference between the real RTT and the estimated RTT for A and B is given by $\rho_{A,B}$, a positive value if the real RTT is larger than the estimate. For a transaction waiting for a log of events (history) from other datacenters with a certain time, the one way latency is affected by a magnitude equal to half the error in the estimation. Thus, the average achieved latency when the effect of the propagation rate and RTT estimation error are factored becomes

$$\bar{L_A} = L_A + Max_{B \in R}\{\theta(A, B) \ + \frac{\rho_{A,B}}{2}\} \tag{4.7}$$

Communication link variability. The observable commit latency of individual transactions differ due to the variability of the communication latency. Thus, the RTT between two datacenters is best represented as a statistical random variable. In this case, Equation 4.7 will observe $\rho_{A,B}$, the error in RTT estimation, as a random variable, $\varrho_{A,B}$, rather than a fixed value, which means that $\bar{L_A}$ is also itself best represented as a random variable.

Compute overhead. The overhead of processing the log and requests in addition to various I/O and interface overheads increase the observed latency. For brevity we call the cumulative effect of all of these overheads, *compute overhead*. Measuring the compute overhead is intractable. However, it effects the observable commit latency in two ways: First, it is the overhead factors of the host of the transaction that we are observing. This can be accounted for in our expression by a random variable, call it $\boldsymbol{C}_{local}$, that affects

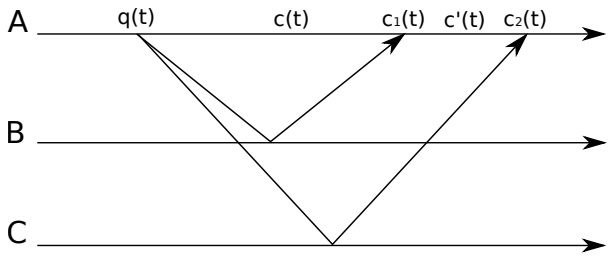


Figure 4.3: A transaction t executing at A demonstrating the trade-off between liveness and commit latency

the observable commit latency directly. Second, it determines the overhead factors that are taking place at other datacenters. These are observed in the commit latency of a transaction as a delay in receiving the needed information. This delay is represented with $\boldsymbol{C}^{B}_{remote}$, where B is the remote datacenter experiencing the overhead. This will make the observable commit latency be

$$\bar{L_A} = L_A + \boldsymbol{C}^{A}_{local} + Max_{B \in R}\{\theta(A,B) \ + \frac{\rho_{A,B}}{2} + \boldsymbol{C}^{B}_{remote}\} \tag{4.8}$$

Bounding observable commit latency. The observable commit latency depends on random variables that are unbounded. This results in the possibility of observing commit latencies that ranges from 0 to an infinitely large number, regardless of the assigned commit latency. Although Equation 4.8 can help us learn the distribution of observable commit latency values, it does not enable us to place an upper-bound or a lower-bound greater than zero on an individual transaction's commit latency. However, since it is a random variable, it will allow us to calculate a probability that a transaction's latency is greater than or equal to a certain value. This, indeed, remains an arduous task, since estimating the compute overhead and link variability random variables is still an area of research.

4.4.2 Latency Trade-off with Liveness and Throughput

Liveness trade-off. There is a trade-off between the observable commit latency and liveness. As the number of tolerated datacenter failures increases, the commit latency is likely to increase as well. To illustrate this consider the scenario in Figure 4.3. A transaction t at A is trying to commit in a scenario with three datacenters. It requests to commit at time $q(t)$. Assume that there are no conflicts and that the knowledge required to commit t is accumulated by time $c(t)$. This means that if we did not want to tolerate failures, and thus use Helios-0, t is able to commit at time $c(t)$. However, if we want to tolerate a single datacenter failure and use Helios-1, t would need to wait until the record of the transaction is acknowledged by another datacenter, which happens to be at time $c_1(t)$. This means that the cost of increasing the level of liveness on t was increasing its commit latency by $(c_1(t) - c(t))$. Likewise, if two datacenter outages are to be tolerated, transaction t commits at time $c_2(t)$ causing an additional penalty on commit latency.

It is not always the case that increasing the number of tolerated datacenter failures will lead to an increase in commit latency. It is possible that the commit latency will remain the same. In Figure 4.3 assume that transaction t receives the sufficient information necessary to commit it at time $c'(t)$. Thus, Helios-0, without tolerating any failures, will commit t at time $c'(t)$. Now, Helios-1, tolerating a single datacenter failure will still commit t at time $c'(t)$. This is because t is already acknowledged by B at time $c'(t)$.

Throughput trade-off. In the chapter, we have focused on minimizing the average commit latency. However, there is an interesting trade-off between average commit latency and throughput. Assigning the optimal commit latency values to Helios will not necessarily result in the best overall throughput. Consider the topology and latency assignment in Table 4.1 of three datacenters with the following RTTs: $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$. The optimal commit latency assignment to

minimize the average commit latency is 5 for A, 25 for B, and 15 for C. Assume that there are no aborts and that transactions complete in exactly the duration of the commit latency. Also, assume that the commit latency values are in ms. For N clients per datacenter, the cumulative throughput when the optimal commit latencies are assigned is: $1000 * N * (\frac{1}{5} + \frac{1}{25} + \frac{1}{15})$, which is $(N * 306.66)$ transactions per second. However, another correct assignment which is 1 for A, 29 for B, and 19 for C results in a cumulative throughput of $(N * 1087.11)$ transactions per second, which is larger than the throughput achieved by the optimal commit latency assignment.

Optimizing for higher throughput can be performed by utilizing the linear programming method in Section 4.2.3. The minimization condition can be changed to be a maximization condition to optimize for the sum of the rate of execution (inverse of commit latency) to be: $\sum_{A \in R} \frac{1}{L_A + c}$, where c is a constant value estimating the execution overhead of a transaction. Introducing c is necessary because otherwise the linear program would assign one of the datacenters a commit latency of 0, which erroneously yields an infinite throughput in the optimization problem. However, in a realistic environment execution takes time and therefore must be accounted for in this case.

More details can be introduced for this optimization to make it more accurate, hence higher throughput. One important factor is contention and the possibility of aborts. The maximization above assumes that all running transactions will commit successfully. However, it is likely that some transactions can abort, and a commit latency assignment that maximizes the expression above for throughput might lead to more contention than other assignments, one of which might have a higher throughput of *committed* transactions.

	V	O	C	I	S
V	-	66 (11)	78 (10)	84 (9)	268 (7)
O	66 (10)	-	19 (1)	175 (7)	210 (4.4)
C	78 (9)	19 (1)	-	175 (7)	182 (6)
I	84 (8)	175 (7)	175 (6)	-	194 (4)
S	268 (6)	210 (4)	182 (6)	194 (4)	-

Table 4.2: RTT latencies between different datacenters in milliseconds and the standard deviation inside parentheses.

4.4.3 Read-only transactions

Read-only transactions in Helios do not contend with other read-write transactions and are performed at the local datacenter. When a Helios instance receives a read-only transaction it chooses a log position to serve as the *snapshot point*. Thus, the read-only transaction will read the state of the data store as of the snapshot point. Every read request, r, will read the data object version that is written by a transaction t, where t is the most recent transaction that writes the data object, r, prior to the snapshot point. This approach is proposed by various log-structured systems such as Hyder [92] and LogBase [93] and is similar to read-only transactions in multi-version databases.

4.5 Evaluation

4.5.1 Evaluation framework

The evaluation of Helios is performed using Amazon's AWS. Helios is evaluated for three different liveness levels: Helios-0 tolerating no datacenter outages, Helios-1 tolerating a single outage, and Helios-2 tolerating two outages. We compare the performance of Helios with Message Futures [7], Replicated Commit [54], and an implementation of Two-Phase Commit (2PC) over Paxos (2PC/Paxos) that is inspired from Spanner [25]. The calculated lower-bound commit latency is also shown for reference.

Objective. The evaluation focuses on quantifying and testing the performance of Helios in addition to its resilience to a lack of strict clock synchronization and RTT estimation errors. The performance metrics we report are commit latency and throughput. These are calculated for transactions that successfully commit. Helios resiliency is tested by artificially reducing the level of synchronization and accuracy of RTT estimation.

Highlights. The highlights of the results of this evaluation are the following:

- Helios-0 achieves a commit latency that is within 54ms from the calculated lower-bound commit latency. This overhead increases as the level of liveness increases (Figure 4.4).
- All variants of Helios achieve lower commit latency and higher throughput compared to Message Futures and 2PC/Paxos. Replicated Commit achieves a lower throughput when compared to Helios but it experiences a commit latency within what Helios-0 and Helios-2 achieve (Figures 4.4 and 4.5).
- Inaccurate synchronization and RTT estimation lead to a higher commit latency (Figure 4.6). The overhead recorded for this set of experiments is up to 52% higher average commit latency.

Setup. Amazon Elastic Compute Cloud (EC2) machines used are High-CPU Extra Large (c3.x2large) which have eight CPU cores and 7GB of RAM memory. Five datacenters were used in the following locations: California (C), Virginia (V), Oregon (O), Ireland (I), and Singapore (S). The average RTT latencies observed are shown in Table 4.2. These RTT numbers are sampled over 24 hours. The standard deviations of the samples are shown inside parentheses. Each datacenter hosts a full replica of the data. One machine in each datacenter runs Helios, serving transactions issued by clients. Helios uses HBase [88] as the underlying data store. Clock synchronization is achieved by running the command "ntpdate ntp.ubuntu.com" before each experiment run.

Workload. Dedicated machines in each datacenter are used to simulate clients and evaluate Helios. We use Transactional YCSB (T-YCSB) [90], a multi-record transactional benchmarking framework, for our evaluations. T-YCSB, an extended version of YCSB [89], generates transactional workloads. It issues transactions that consist of a set of read and write operations, where each operation accesses a different record of the data store. Each client can have one outstanding transaction at a time. Clients issue transactions as fast as they can unless mentioned otherwise. An operation is either a read or a write to a key from a pool of 50000 keys. The key is chosen using a Zipfian distribution. Each transaction contains five operations. Half of these operations are reads and the other half are writes. Each experiment runs for a duration of 10 minutes.

Configuration. Helios assigns the commit offsets to target achieving the lower-bound commit latency. The RTT average values in Table 4.2 are used to derive the optimal commit latencies using the method shown in Section 4.2.3. The calculated optimal commit latencies are: 69ms, 10ms, 10ms, 166ms, and 200ms for V, O, C, I, and S, respectively. These are then used to derive commit offsets to be used to commit transactions in Helios using Equation 4.5. As a baseline, we also display results of Helios without performing RTT estimations and optimal latency calculations by setting all the commit offsets to 0. This baseline is called **Helios-B**.

4.5.2 Systems for comparison

We report Helios results compared with results obtained from Message Futures [7], Replicated Commit [54], and a Two-Phase Commit over Paxos protocol (2PC/Paxos) that is inspired by Spanner [25].

Message Futures, like Helios, uses replicated logs that are causally ordered to exchange transactions information and detect conflicts. In Message Futures, partial logs

are continuously being propagated between datacenters [87]. Transactions, t^i, which request to commit at A between the transmissions of partial log i and $i+1$ are assigned a reservation number i. Partial log transmissions are acknowledged by other datacenters. The acknowledgment contains a log of all transactions (not known to A) up to the time, τ, of the transmission of the acknowledgment. Transactions t^i can commit when log i is acknowledged by all other datacenters and no conflicts are detected. Helios follows a different approach where a transaction commits by waiting for partial logs from other datacenters up to calculated timestamps that minimizes the commit latency while preserving correctness.

Replicated Commit uses Paxos for cross datacenter replication and Two-Phase Locking (2PL) to avoid conflicts. Clients first perform read operations by trying to lock the read object in a majority of datacenters. Write operations are buffered. Then, committing the transaction is done by replicating it using Paxos to all datacenters. As the transaction is received by datacenters, locks are acquired for buffered write operations and read locks are validated. If the locks were acquired and validated at a majority of datacenters, then the client will commit the transaction. Thus, its commit latency should be in the order of a single RTT to the closest majority, which is the time required by Helios-2 to commit. However, Replicated Commit performs poorly in terms of throughput due to the read strategy. The client spends more time reading before the commit request is issued.

2PC/Paxos uses 2PC to avoid conflicts and Paxos for replication. In this evaluation the datacenter in Virginia (V) is assigned to be the 2PC coordinator. Clients, scattered across all five datacenters issue reads to the coordinator. The coordinator maintains a lock table. When a read is received, a read lock is placed on the corresponding key. Write operations are buffered. When a transaction is ready, a request to commit is sent to the coordinator with information of all operations. Once the coordinator receives the commit request it tries to acquire the write locks and verifies that the read locks are

still held. If successful, the transaction commits. Then, the coordinator replicates the log to a majority of datacenters using Paxos [26]. The coordinator is assumed to have a lease so that it will not need to go through the leader election phase, hence reducing the required time to replicate the log by one RTT. After replicating the log, the output of the transaction is sent to the client. Transactions that are detected to be involved in a deadlock are immediately aborted.

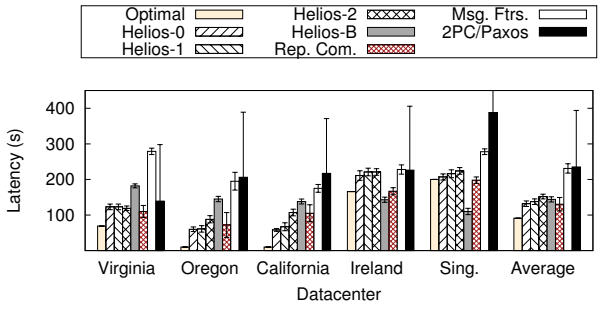
Replicated Commit and 2PC/Paxos replicate to a majority before committing a transaction. Thus, in the setup used for the following experiments with five datacenters, they tolerate two datacenter failures. Thus, they tolerate the same number of failures as Helios-2. Message Futures does not inherently overcome datacenter outages, thus its liveness guarantee is equivalent to Helios-0.

Like Helios-2, Replicated Commit and 2PC/Paxos can operate as long as a majority of datacenters are operational. Message Futures does not inherently overcome datacenter outages, thus its liveness guarantee is equivalent to Helios-0.

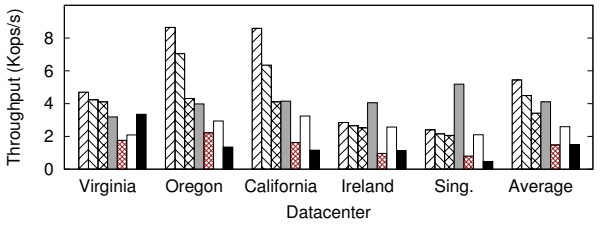
4.5.3 Helios performance

In the following experiments, 60 clients scattered across all datacenters issue back-to-back transactions to Helios instances. We report the results for each datacenter separately in addition to confidence intervals for the commit latency numbers.

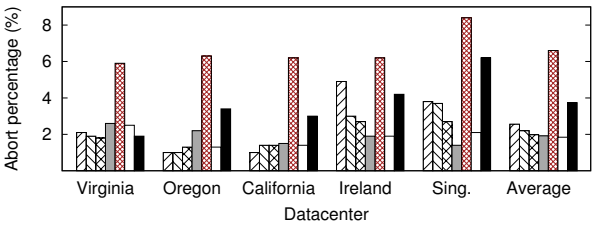
Latency. The commit latency results are shown in Figure 4.4(a) for Helios with varying liveness levels. We also show the commit latency of the baseline systems. In the figure, the average latency of clients that are running at each datacenter is shown. Helios-0 has the closest latencies to the optimal since it does not need to synchronously replicate any transactions before committing them. The overhead of Helios-0 over the optimal commit latency is within 7 to 54ms. This overhead is caused by different factors



(a) Commit latency of clients at each datacenter.



(b) Throughput at each datacenter.



(c) Abort rates.

Figure 4.4: The commit latency, throughput, and abort rate of a scenario with 60 clients and 5 datacenters.

such as execution time, I/O access, network variability, and the network latency between the client and server. Some of these factors are analyzed in Section 4.4.

Increasing the level of fault-tolerance results in an increase in commit latency (see Section 4.4.2 for more details). This increase is caused by the difference between the commit latency needed to commit consistently and the required time to get the acknowledgment of the transaction's reception by the number of tolerated datacenter failures. This difference can vary, which explains why the overhead of tolerating one datacenter failure by using

Helios-1 to Helios-0 ranges from 0-1ms for Virginia and Oregon while the overhead is 9-10ms for the remaining datacenters. Virginia and Oregon do not experience a significant overhead because their commit latencies are equal to or greater than the RTT to the closest datacenter, thus receiving the acknowledgment of the replication happens while waiting for transaction commitment. Similar behavior is observed when increasing the number of tolerated datacenter failures from 1 to 2, which results in an increase of commit latency ranging from no observed overhead for Virginia, Ireland, and Singapore to 27ms and 40ms overhead for Oregon and California. The average commit latency for Helios-2 at Virginia is actually 4ms lower than the latency of Helios-1. This does not mean that less time is required to commit for Helios-2, rather it is due to the variability of the compute and network conditions.

Replicated Commit commits transactions with a latency equal to the closest RTT majority. The achieved average latency is 20ms lower than the one achieved by Helios-2. This is because Helios experiences more compute overhead to process and send the log, while Replicated Commit's communication process is lightweight. The average commit latency of Replicated Commit is also close to Helios-0. The evaluation topology and RTT values for this set of experiments make the overhead for tolerating two datacenter outages not so large. Coupled with the higher compute overhead of Helios, this results in observing similar commit latency for Helios-0 and Replicated Commit. However, in topologies where the overhead to tolerate two datacenter outages is higher, Helios-0 should show a more significant advantage over Replicated Commit.

Message Futures requires roughly a RTT to all other datacenters to commit transactions. This causes an overhead compared to Helios-0 that ranges from 17ms for Ireland to 181ms for Singapore. The average overhead is 99ms. 2PC/Paxos requires more time to commit when compared to Helios-2, which has the same degree of fault tolerance. The difference in commit latency between 2PC/Paxos and Helios-2 ranges from 15-17ms for

Virginia, Ireland, and Singapore to 146-159ms for Oregon and California. In 2PC/Paxos, a transaction experiences a latency equal to the time required to send the transaction to Virginia in addition to a RTT to a majority from Virginia. This gives an advantage to Virginia and datacenters close to it. Helios-B, which is Helios-0 but without assigning optimal commit latencies, has an average overhead of 12.2ms compared to Helios-0.

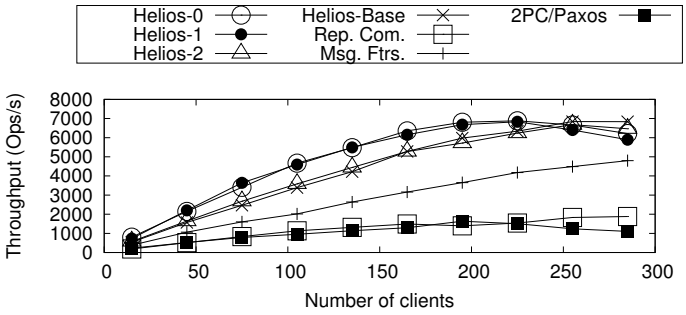
Helios and Message Futures are stable with standard deviation values that are less than 10. Replicated Commit and 2PC/Paxos on the other hand show unstable performance with standard deviation values of up to 35 for Replicated Commit and values ranging from 154 to 278 for 2PC/Paxos. The stability observed for Helios and Message Futures is due to the use of a log to exchange transactions between datacenters. If a transmission i was affected by the variability of the communication network, the next transmission, $i + 1$, will include i because it is part of the log that was not acknowledged [87].

Throughput. The throughput results are presented in Figure 4.4(b). Helios-2 achieves a throughput 37% lower than what is achieved by Helios-0. The throughput achieved by Replicated Commit, Message Futures, and 2PC/Paxos is lower than what is achieved by Helios. The commit latency causes this lower throughput for Message Futures. Replicated Commit and 2PC/Paxos have a larger overhead due to their read strategy. Replicated Commit reads from a majority and 2PC/Paxos directs reads to Virginia which increases the amount of time spent by a client prior to requesting to commit. Remember that the commit latency is the time from the client's commit request till a decision is received and does not include the read latency incurred prior to the commit request. Replicated Commit and 2PC/Paxos achieve an average throughput that is 56-57% lower than Helios-2. Note that Virginia in 2PC/Paxos achieves the closest throughput to Helios-2 although it was not the closest to Helios in terms of commit latency. This illustrates the advantage of 2PC/Paxos clients at the master datacenter ridding them from the overhead of wide-area reads. Message Futures's throughput is 52%

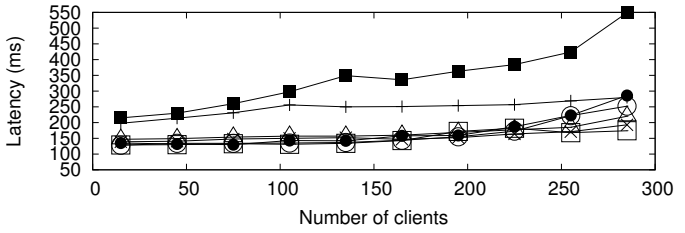
lower than the throughput of Helios-0. For Message Futures, the achieved throughput by each datacenter compared to Helios is directly correlated to the overhead in commit latency. This is because Message Futures, like Helios, reads from the closest datacenter. Helios-B achieves a throughput that is 24% lower than Helios-0. Like Message Futures, the throughput overhead correlates directly with the latency overhead.

Contention. The abort rates are shown in Figure 4.4(c). The abort rate is a product of many factors, such as the amount of contention, the number of concurrent transactions, the lifetime of the transaction, among others. However, we can observe a pattern in the average abort rates of different systems. Increasing the liveness level of Helios causes the abort rate to decrease, which is a sign of less contention that results from the decrease in throughput. Message Futures has the lowest abort rate although it has a higher commit latency compared to Helios. Replicated commit and 2PC/Paxos on the other hand achieve worse abort rates. 2PC/Paxos has the highest commit latency and additionally holds locks for an extended period of time, beginning from the read operations prior to the commit request until releasing locks after the commit phase. This increases contention and is a factor causing this high abort rate. Replicated Commit locks data object in lock tables scattered across datacenters which increases the chance of experiencing deadlocks. Abort rates of individual datacenters illuminate how having a larger commit latency is a disadvantage causing more transactions to be aborted; Singapore experiences larger abort rates when compared to other datacenters.

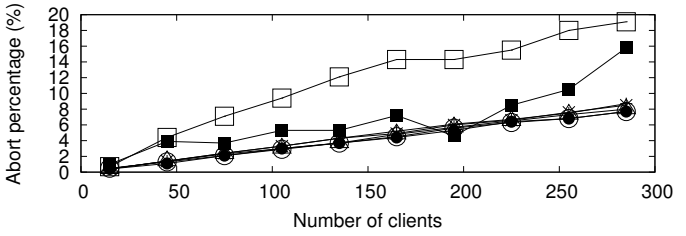
Peak throughput. Now, we measure the peak achievable throughput and the number of clients required to converge to that throughput for Helios. We increase the load on the system by gradually increasing the number of clients issuing transactions. In Figure 4.5(a), the cumulative achievable throughput is plotted with the number of clients in the system increasing from 15 clients to up to 285 clients in 30-clients increments. Note how the Helios protocols converge to a throughput between 6000 and 7000 operations per second.



(a) Throughput as the number of clients increases.



(b) Commit latency as the number of clients increases.



(c) Abort rates.

Figure 4.5: The throughput, average latency, and abort rate as the number of clients is increased.

Helios-0 and Helios-1 are the fastest to converge as soon as the number of clients is 195. Helios-2 and Helios-B converge with 255 clients. Message Futures takes a slower pace than Helios. This is mainly due to its higher commit latency values. 2PC/Paxos is not able to achieve a throughput that is larger than 1700 operations per second. The demand placed on the coordinator causes thrashing as soon as the number of clients reaches 195 clients. Replicated Commit experiences a similar throughput to 2PC/Paxos but does not thrash.

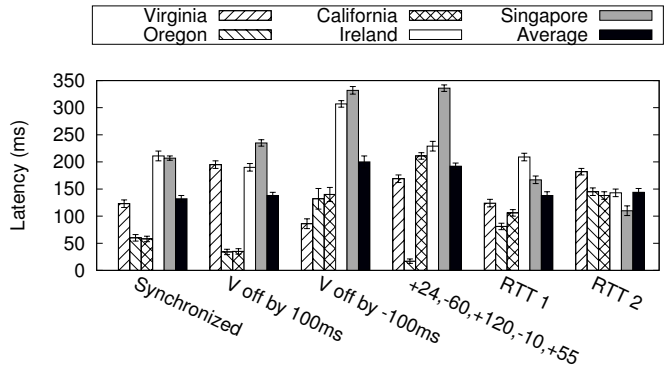
Converging to the peak throughput signals an I/O bottleneck. Increasing the demand

(number of clients) past the convergence point stresses the bottleneck, causing a gradual degradation of the performance of individual transactions. Figure 4.5(b) shows the effect of increasing the number of clients on the commit latency. Helios variants maintain their commit latency up to the convergence point between 195 and 255 clients. Message Futures and Replicated Commit maintain their commit latency because they need more clients than what is shown for them to converge. 2PC/Paxos starts a gradual increase in commit latency from the beginning signaling a stress on the system even with a small number of clients. Placing a large demand on a single coordinator causes resources to be exhausted rapidly, leading to this observation on 2PC/Paxos.

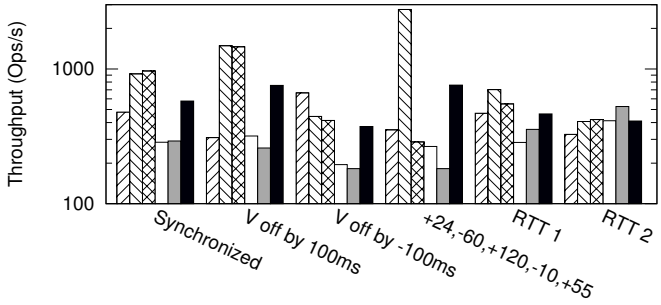
Increasing the number of clients increases the contention and leads to more aborts. Figure 4.5(c) shows this effect. All protocols except Replicated Commit and 2PC/Paxos have similar abort rates that are increasing around 0.7% for every 30 clients added. 2PC/Paxos abort rate increases significantly for 258 clients to have 15.8% aborts, while Replicated Commit experiences close to 20% aborts for 258 clients.

4.5.4 Synchronization and estimation errors

Synchronization. The performance of the Helios protocols relies heavily on the level of synchronization. Using the readily available NTP clients provided by Ubuntu for the previous experiments shows that the available synchronization tools enable Helios to achieve good performance. However, we would like to get an insight on Helios performance with more hostile environments with clocks that are not synchronized. Now we will show results from a set of experiments while changing the clock readings of machines to emulate a lack of synchronization. In these experiments Helios-0 is used. The results are shown in the leftmost four groups of results in Figure 4.6(a). In the figure we display three experiments and compare them with the results from a synchronized run using NTP. The



(a) The effect on commit latency.



(b) The effect on throughput.

Figure 4.6: The effect of the lack of synchronization on the performance of Helios-0 in the leftmost four groups of results and the effect of erroneous RTT estimation in the rightmost two groups of results.

first non-synchronized scenario is to set the clock at Virginia ahead of the others by 100ms. Note how this resulted in its commit latency to increase by 62ms. Other datacenters commit latencies improved except for Singapore. This improvement is caused by setting Virginia's clock ahead in time, it puts it in a disadvantage compared to other datacenters which leads to a better commit latency for others. The second scenario has Virginia's clock 100ms behind other clocks. This change puts Virginia in an advantage that is shown by a decrease in commit latency of 37ms. However, other datacenters consequently achieve a higher commit latency and cause an increase in the average commit latency by 64ms compared to the synchronized case. In the final scenario we introduce random errors

to all datacenters. The errors are presented by a sequence of numbers denoting the shift in time for the ordered set of datacenters in milliseconds: V, O, C, I, and S. Consider the case {+24,-60,+120,-10,+55}. We find that for this case the average commit latency is 60ms higher than the synchronized case. However, for some individual datacenters, this caused a significant improvement of the commit latency; California, for example, achieves a commit latency 23ms lower than California in the synchronized case.

The results demonstrate that small to medium clock skews introduce tolerable increases in the average commit latency of up to 64ms for the scenarios we considered. These increases are correlated with the level of synchronization.

RTT estimation errors. A vital part of Helios is the assignment of optimal commit latencies to datacenters. However, deriving these values requires estimating the RTT between datacenters. RTTs can be variable and change occasionally. Thus, the calculated assigned commit latencies suffer from the potential of being inaccurate. Here we perform tests to quantify the effect of non-accurate estimations of the RTT on the commit latency of Helios. Helios-0 will also be used for these experiments. We will experiment with two cases where high margins of error are introduced to our RTT estimations. The results are shown in Figure 4.6(a) labeled RTT estimation 1 and 2. The first RTT estimation is calculated by introducing an error to the correct estimates by increasing one fifth of the RTTs by 25ms, another fifth by 75ms, decrease a fifth of RTTs by 25ms, and decrease yet another fifth by 75ms, and the remaining RTTs are not changed. This scenario leads to the average latency being 4.5% higher that the one achieved by the original estimate. The second RTT estimate is of an erroneous estimation of no latency between datacenters. This will lead to assigning all datacenters a commit latency of 0. With this input, Helios-0 observes the following commit latencies for V, O, C, I, and S: 182ms, 145ms, 138ms, 143ms, 110ms, averaging to 144ms. This average is 9% higher than the average commit latency achieved by the original estimate.

These results illustrate that even with highly erroneous RTT estimations, Helios-0 observes a slight increase in the achieved commit latencies by 4.5% and 9% for the introduced errors. Helios shows the same stability observed in previous experiments where standard deviation values are less than 10.

Throughput. When errors are introduced to clock synchronization and RTT estimation, the commit latency is affected for different datacenters. Some datacenters have their commit latency increase while for others their commit latency decrease. The commit latency of a datacenter is a major factor of the observed throughput. Thus, we expect different datacenter throughputs to be affected in correlation with the commit latency. We present the achieved throughput results in Figure 4.6(b). For the scenario when Virginia's clock is 100ms ahead of others, notice how the throughput of Oregon and California increase as a result of the decrease in their commit latency. Likewise, the throughput of Virginia decreases in correlation to the commit latency's increase. This effect is observed throughout the set of experiments. However, an interesting observation is that for some scenarios, this leads to the average throughput to become larger than the synchronized case. For the *+24,-60,+120,-10,+55* case, for example, throughput is 31% higher than the synchronized case. This is even though the average commit latency of *+24,-60,+120,-10,+55* is higher than the original case. The increase in throughput is due to the extremely low resulting latency of Oregon allowing it to achieve a throughput of 2764 operations/s that caused the average throughput to exceed that of the synchronized case. We provide more discussion on this trade-off in Section 4.4.2.

4.6 Conclusion

In this chapter, we have formulated a lower-bound on commit latency of transactions running on replicated data stores. We also designed an optimistic commit protocol,

called Helios, using insights from the lower-bound. Helios allows tunable performance for individual datacenters. In the chapter, we demonstrate tuning Helios to minimize the average commit latency. Helios separates consistency from liveness guarantees. This allows the flexibility of setting the number of tolerated datacenter outages. Helios is evaluated experimentally and compared to three systems: Replicated Commit, Message Futures, and 2PC/Paxos. Helios outperforms the other systems in the study and approaches the optimal transaction latency.

Part II

Global-Scale Analytics and Learning

Chapter 5

Ogre: Efficient Analytics on Global-Scale Data

5.1 Introduction

Many essential tasks for web and cloud applications involve reading a large portion of stored data. An example is real-time data analytics that is increasingly being used to drive business decisions. Another example is data maintenance, which includes tasks like rebuilding indexes and creating snapshots for migration. Since analytics and maintenance tasks read a large portion of stored data, processing these tasks as general read-modify transactions is prohibitive. This is because the large size and longevity of these tasks lead to an influx of aborts due to conflicts. This motivates a plethora of research work on designing commit protocols and techniques specific to *read-only transactions* that are capable of efficiently handling analytics and maintenance tasks [19, 20, 24, 25].

Read-only transaction techniques have different designs and performance characteristics. For geo-replication, two characteristics are essential for read-only transactions: (1) Read-only transactions execute and commit locally in the datacenter, *i.e.*, without

cross-datacenter communication, and (2) Read-only transactions do not conflict with on-going read-modify transactions. We will call transactions that satisfy these two properties **efficient read-only transactions**. These two properties *cannot* be satisfied for arbitrary read-modify transactions since conflict is inevitable and coordination is necessary to detect conflicts.

Existing protocols for read-only transactions have been successful in ensuring one or both properties of efficient read-only transactions but at the cost of increasing the complexity of read-modify transactions. A read-only transaction can commit locally without communicating with other replicas by ensuring that the local replica's state is consistent independent of other replicas [19]. Ensuring the consistency of a replica can be achieved by guaranteeing that the order of transactions applied to each replica is a consistent order. This is typically attained by shipping ordered logs or gossip to communicate transactions information [24]. A common method to avoid conflicts between read-only transactions and ongoing read-modify transactions is multi-versioning; a read-only transaction reads a *historical* consistent snapshot of the multi-versioned store [20].

Supporting both properties of efficient read-only transactions is possible with a combination of ordered log shipping and multi-versioning. However, these two techniques can be harmful to coexisting read-modify transactions. Transmitting transaction information across datacenters using an ordered log is not scalable due to the single-point of contention in the head-of-the-log. Likewise, multi-versioning is harmful to read-modify transactions because it increases memory footprint and puts more pressure on garbage collection of historical data.

In this chapter, we propose *Ogre* (*Optimistic Geo-REplication*), an optimistic transaction commit protocol for geo-replication. An essential motivation of Ogre's design is optimizing the performance of read-only transactions without significantly affecting the

performance of coexisting read-modify transactions. Ogre supports efficient read-only transactions in a novel protocol that overcomes the limitations of existing systems. Most notably, Ogre does *not* use an ordered log to propagate transaction information. Shipping ordered logs is the main vehicle for traditional designs of read-only transactions on replicated systems. However, ordered logs are detrimental to performance and scalability due to their sequential nature and contention on the head of the log. Instead, Ogre employs a dependency tracking mechanism to apply transactions in a consistent order. *Dependency tracking scales beyond what is possible with log shipping because it only enforces the ordering constraints that are necessary for consistency, rather than enforcing a (log's) total order which is not necessary.*

Ogre cleanly separates between the storage used to commit read-modify transactions (current storage) and the storage used to serve read-only transactions (multi-versioned storage). This property, called *analytics isolation*, reduces the overhead of multi-versioning on current storage. In addition, this separation allows the scaling of read-only transactions by hosting multiple multi-versioned storage units possibly in different machines for each replica. Additionally, each one of those multi-versioned storage units may be responsible for a different type of analytics workload.

Ogre's commit protocol for read-modify transactions is a variation of a majority protocol [17] that is extended to support dependency tracking for read-only transactions and analytics isolation for storage. Read-modify transactions read locally and commit from a majority. This design ensures competitive performance for read-modify transactions compared to current geo-replication systems [37, 33, 94, 25, 53, 54, 7, 8], while not significantly affecting the staleness of read-only transactions. Additionally, unlike existing geo-replicated efficient read-only transaction designs [25], Ogre does not require any special hardware support in the form of strictly synchronous clocks.

The rest of this chapter begins with the design of Ogre in Section 5.2 and its correctness

proofs in Section 5.3. Section 5.4 reports the results of an experimental evaluation on five datacenters. Finally, we conclude in Section 5.5.

5.2 System design

In this section, we propose *Ogre*, an optimistic transaction commit protocol for transactions on geo-replicated data. Ogre incorporates *dependency tracking* and *analytics isolation* to support efficient read-only transactions. We begin with a high-level overview of Ogre, followed by more details about its architecture and algorithms.

5.2.1 Overview

Ogre executes transactions on replicas of data dispersed over multiple datacenters. Each datacenter hosts a full replica of the data. We use the terms datacenter and replica interchangeably to denote the full replica at a datacenter. Ogre resembles traditional majority protocols, but it extends them to support efficient read-only transactions. We begin by describing the part of Ogre that resembles traditional majority protocols (Section 5.2.1). Then, we explain why this basic majority protocol cannot provide efficient read-only transactions (Section 5.2.1). Finally, we propose our dependency tracking mechanism and analytics isolation to support efficient read-only transactions (Section 5.2.1).

Design basis

Ogre differentiates between two types of transactions: (1) *Read-modify transactions*, which consist of reads and at least a single write, and (2) *Read-only transactions*, which

consist of read operations only[1]. Ogre commits read-modify transactions optimistically, meaning that an application client (*client* for short) executes a transaction by reading from local storage and buffering writes. Once all operations are performed, the transaction's information is sent to *all* datacenters to collect their votes. This communication is called a *prepare message*. Replicas validate (also called *certify*) the transaction and reply with a positive or negative vote. A replica replies to a prepare message for a transaction T with a positive or negative vote according to a *voting rule*. The voting rule is the following:

Definition 2 ***(Majority voting rule)***. *A replica R replies with a positive vote for a transaction T if and only if:*

- *(1) read versions by T are current (not overwritten) in replica R, and*
- *(2) transaction T does not conflict with* pending *transactions at replica R. A pending transaction is one that is verified by replica R but a commit/abort decision for it is not received yet.*

A positive *majority* vote commits the transaction. Otherwise, the transaction aborts. After the client gets a majority vote, an *apply message* is sent from the client to all replicas to announce the outcome (*i.e.*, commit or abort decision) of the transaction.

Inefficient read-only transactions

Up to this point, Ogre resembles traditional majority protocols, which do not have special support for read-only transactions. Ogre extends majority protocols to support read-only transactions with the following properties, called *properties of efficient read-only transactions*:

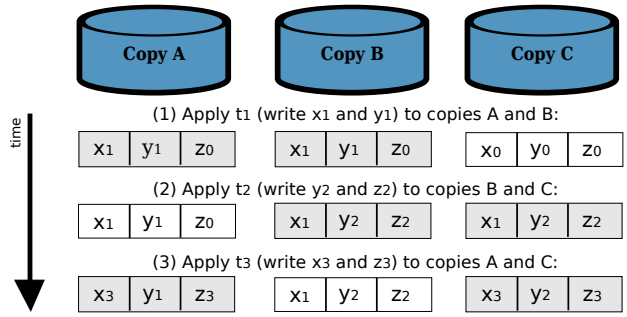
Definition 3 *The properties of efficient read-only transactions are:*

[1]In the remainder of this chapter, we often refer to read-modify transactions as transactions for short. We will always refer to read-only transactions by the full term.

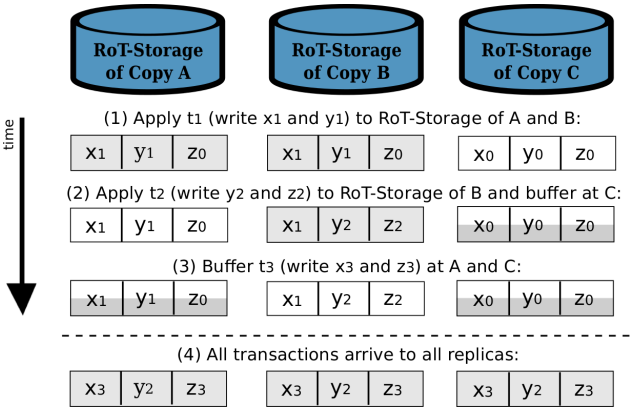
- *(1) Read-only transactions do not conflict or interfere with ongoing read-modify transactions, and*
- *(2) Read-only transactions are served from the local replica without incurring any wide-area latency.*

Simple majority protocols cannot support efficient read-only transactions. This is because reading the state of one replica is *not* guaranteed to produce a consistent view of the Serializable history. Traditional majority protocols, when committing read-modify transactions, overcome this by verifying that the reads are current at a majority in the *prepare* phase when collecting votes. Collecting votes require cross datacenter communication. However, we would like to avoid cross datacenter communication and serve read-only transactions locally.

To illustrate why simple majority protocols need to consult a majority to validate a read-only transaction, consider the following scenario of three datacenters A, B, and C, depicted in Figure 5.1(a). Each row in the figure depicts the state of all replicas after a transaction commits. For example, the first row represents the state after transaction t_1 commits and writes x_1 and y_1 on copies A and B. The state consists of the values of three data objects, x, y, and z. Object x_i is a version of x that was written by transaction t_i, where x_0 is the initial version. A shaded state denotes that the transaction commits at that replica. First, transaction t_1, which writes x_1 and y_1, commits in datacenters A and B. Assume that t_1's information did not reach C. Then, t_2, which writes y_2 and z_2, commits in datacenters B and C. Also, assume that t_2's information did not reach A. Next, in step 3 transaction t_3, which writes x_3 and z_3, commits at A and C. This time, assume that B did not receive t_3. At this point, the versions at A are $[x_3, y_1, z_3]$, at B are $[x_1, y_2, z_2]$, and at C are $[x_3, y_2, z_3]$. Although datacenters B and C have a consistent



(a) A traditional majority protocol does not guarantee that the state of any single replica is consistent



(b) Ogre delays incorporating transactions in RoT-Storage until their dependencies are met which guarantees that RoT-Storage has a consistent snapshot. Ogre's current storage is identical to traditional majority protocols.

Figure 5.1: A scenario depicting state changes in traditional majority protocols and Ogre

snapshot[2] of the data objects, this is not the case for datacenter A. This is because copy A misses writes of t_2, which is a dependency of t_3. Thus, *a read-only transaction that reads the state of datacenter A after step 3 will return an incorrect snapshot.*

Existing solutions to support efficient read-only transactions leverage techniques such as *ordered log shipping* and *multi-versioning* [20, 24]. Ordered log shipping is used to keep each replica in a consistent state by ordering transactions in the log in an order that preserves their dependencies. However, preserving a log's total order introduces an

[2] B reflects execution of t_1 and t_2, and C reflects execution of t_1, t_2, and t_3.

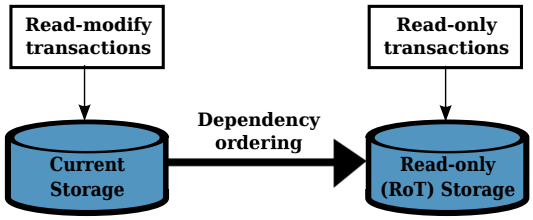


Figure 5.2: The components of an Ogre replica demonstrating analytics isolation

overhead that limits scalability. Multi-versioning allows a read-only transaction to read as-of a past time so that it does not conflict with ongoing read-modify transactions. However, maintaining historical versions introduces an overhead on the access performance to current data, which affects read-modify transactions. Recently, Spanner [25] proposes the use of accurate time synchronization to support read-only transactions. This, however, requires a special infrastructure (*i.e.*, atomic clocks) that is not accessible to all applications.

The gist of Ogre

Ogre supports efficient read-only transactions by ordering transactions and multi-versioning, but in a different way from existing solutions. Ogre does not enforce order via an ordered log. Rather, it employs a dependency tracking mechanism to enforce order only between dependent transactions. This helps in reducing the overhead caused by ordered log shipping in existing solutions. Ogre also isolates the historical storage used for read-only transactions from the current storage used for read-modify transactions, called analytics isolation. This way, the overhead of maintaining historical versions will not directly influence the access time of current versions. Ogre's isolated multi-versioned copy of storage is called the *Read-only Transactions Storage* (RoT-Storage), which coexists with the *current storage*. Figure 5.2 shows the components of a replica. Read-modify transactions are sent to the current storage. The current storage forwards committed transactions information to RoT-Storage. A transaction is applied to RoT-Storage only

after its dependencies are satisfied. It is possible to have multiple RoT-Storage copies for each replica that exist on different machines.

Ogre guarantees that RoT-Storage has a consistent view of the Serializable history by enforcing the following invariant: *A committed transaction's write-set is applied to RoT-Storage only after its dependencies are applied.* Thus, if a transaction T's commit message is received by a replica and it depends on a transaction T' with a commit message that has not been received yet, the replica will buffer T (T will be applied to the current storage but not yet to the RoT-Storage). Once T' is received by the replica, T will be applied after T'.

In Figure 5.1(a) we showed an example of a simple majority protocol that leads to an inconsistent replica state for local read-only transactions. We now apply the same scenario on Ogre and show the replicas state in Figure 5.1(b). We do not show Ogre's current storage in the figure as it is identical to the storage of the simple majority protocol in steps 1, 2, and 3 shown in Fogire 5.1(a). Ogre's RoT-storage, however, is different and is the one shown in Figure 5.1(b). In the Figure, a state that only has its bottom half shaded denotes that a transaction is received but is buffered, waiting for unmet dependencies. The difference in RoT-Storage is that it buffers t_3 at A, and buffers t_2 and t_3 at C. Thus, by the third step in the scenario, the versions at A's RoT-Storage become $[x_1, y_1, z_0]$, at B become $[x_1, y_2, z_2]$, and at C become $[x_0, y_0, z_0]$. At any point, a read-only transaction can read a consistent, albeit stale, snapshot from any single replica's RoT-Storage. Ogre propagates transactions information to *all* replicas. Thus, eventually, each replica receives all transactions. After all transactions are received at all replicas (step 4), RoT-Storage applies the buffered transactions and the versions in all replicas (RoT-Storage and current storage) become $[x_3, y_2, z_3]$.

To decide whether to apply or buffer a transaction at RoT-Storage, Ogre needs to identify the dependencies of a transaction T upon receiving it. For this, we design

the application client's algorithm to ask for dependencies in the prepare message. The replica in the vote message, rather than replying with a vote only, also sends information about transaction dependencies. There are three types of dependencies: (1) *read-from dependencies:* transactions that wrote the data objects read in the read-set, (2) *write-write dependencies:* transactions that wrote the most recent versions of each object in T's write-set, and (3) *read-before dependencies:* transactions that read the most recent versions of each object in T's write-set. By checking these dependencies, Ogre decides whether a transaction should be applied or buffered.

In the remainder of this section, we present Ogre's algorithms to commit transactions, track dependencies, and manage current and RoT-Storage.

5.2.2 Ogre algorithms

Ogre's algorithms are run by application clients and replicas. A client drives transaction execution and commitment. It sends requests to replicas to validate and commit transactions. We make the following assumptions: (1) The communication channel is a FIFO channel. Thus, an underlying ordered messaging protocol (*e.g.*, TCP) is used, (2) Failures are fail-stop, and (3) A client persists a transaction's information locally before execution. Thus, in the case of a datacenter-scale outage, a client is able to carry out transaction execution after recovery.

We begin by showing the application client's algorithm for read-modify transactions (Section 5.2.2), replica's algorithm to serve read-modify transactions from the current storage (Section 5.2.2), then we present the algorithms for dependency tracking in RoT-Storage and the efficient execution of read-only transactions (Section 5.2.2).

Algorithm 6: Ogre's algorithm to drive read-modify transactions execution by an application client

```
1: On receiving a transaction to commit {
2: in: A transaction T
3: T.Id := uniqueId
4: decision := NULL
5: for each operation o in T do
6:     if o is a write then
7:         Buffer o in T.writeset
8:         o.version = -1
9:     else // read operation
10:        // read from the closest datacenter
11:        value, version ← read(from: local replica, object: o)
12:        add {o, value, version} to T.readset
13: Send(to: replicas, type: prepare_request, body: T)
14: while decision is NULL do
15:     vote, write_versions, rb_deps ← receive_next_vote()
16:     if vote is positive then
17:         commit_votes++
18:         for each operation o_w in T.writeset do
19:             if write_versions[o_w] > o_w.version then
20:                 o_w.version = write_versions[o_w]
21:                 T.rb_deps[o_w] = rb_deps[o_w]
22:             if write_versions[o_w] == o_w.version then
23:                 T.rb_deps[o_w] = T.rb_deps[o_w] ∪ rb_deps[o_w]
24:     else
25:         abort_votes++
26:     if commit_votes is a majority then
27:         decision = commit
28:     else if abort_votes is a majority then
29:         decision = abort
30: Broadcast(to: replicas, type: decision, body: T)
31: }
```

Application client protocol for read-modify transactions

Application clients drive the execution of transactions. Algorithm 6 shows the algorithm to commit a transaction and collect the transaction's dependencies for later incorporation to RoT-Storage.

Transaction structure. A transaction, T consists of a unique id, a binary commit-

ment decision, and a read and write-sets. Each operation in the read-set has the key, value, and read version of the data object. Each operation in the write-set has the key, value, write-version, and read-before dependencies (*rb_deps*). The write-version of a write object o, $o.version$, denotes the largest written version of o from the collected votes. Read-before dependencies of o, are the ids of *read-modify transactions* which have read the version $o.version$ of the data object. Read versions, Write versions, and read-before dependencies are collected from votes and are sent with the commit message to all replicas. These are the dependency information that controls when a transaction is applied to RoT-Storage.

Initialization and executing operations. When a transaction begins, a unique Id is assigned (Line 3). The application client then executes all operations in the transaction's read and write-sets (lines 5-12). Write operations are buffered locally at the client and initialized by setting their version value to -1. The write version numbers are going to be updated while collecting the votes. Reads are served from the local replica. The client gets the value and version number of each read object. The version number is a monotonically increasing number denoting the number of overwrites of a data object.

Commit protocol. After executing operations, the client asynchronously sends a *prepare request* to all replicas (line 13). The client then collects votes until a majority of votes is received (lines 14-29). A *positive* vote from a datacenter R is accompanied with the following dependency information: (1) *write_versions*: current version numbers of the objects in T's write-set at R, and (2) *rb_deps*: read-before dependencies for each object in T's write-set at R. In the case of receiving a positive vote, a counter of positive votes is incremented (line 17). Then, the write-set are updated according to the received dependency information (lines 18-23). The information of a data object in the write-set is updated if the received write version of that object is larger than or equal to what is previously observed (lines 19-23). In that case, the version of the data object is updated to

become equivalent to the largest version received with the votes (line 20). If the received write version is *larger* than what is previously observed, then the list of read-before dependencies of the data object o_w, denoted $T.rb\_deps[o_w]$, is replaced with the one received. This is because the previous read dependencies correspond to an earlier version, and thus must be replaced (line 21). If the received write version is *equal* to what is previously observed, then the existing read dependencies and received read dependencies are combined, because they are all valid for that particular write version (line 23). A negative vote will result in incrementing the counter of negative votes (line 25). According to the collected votes, the transaction will send either a commit or abort message to all replicas accompanied with the transaction's information (lines 26-30).

Current storage protocol for read-modify transactions

A replica's current storage receives prepare, commit, and abort requests from clients executing read-modify transactions. Algorithm 7 shows how these requests are handled in Ogre's current storage.

Replica state. Each replica maintains the following meta-data in the current storage (lines 1-3): (1) *Pendings*: a list of validated transactions that are not committed or aborted, (2) *Readers*: a map from data objects to lists of transaction Ids called readers of a data object. Readers of a data object x are committed transactions that have read the current version of data object x.

Prepare requests. A prepare message is a request from a client to a replica to vote on a transaction T (lines 5-14). A replica applies the voting rule (Definition 2) to decide whether to vote positively or negatively.

The prepare request contains information about transaction T's read and write-sets. First, conflicts are detected between T and pending transactions using the function detect-conflicts() shown in lines 34-40. A conflict is an intersection between transaction

Algorithm 7: Algorithms to process read-modify transaction requests at Ogre's current storage.

```
1: STATE
2:     Readers: A map from data objects to lists of transaction Ids, initially each list is empty
3:     Pendings: A list of transactions, initially empty
4:
5: On receiving a prepare request {
6:     in: Transaction T
7:     if detect-conflicts(T, Pendings) is reject OR verify-read-versions(T) is reject then
8:         reply(vote: negative); exit
9:     Pendings.add(T)
10:    // Collect dependencies
11:    rb_deps = get-rb-transactions(T)
12:    write_versions = get-write-versions(T)
13:    reply(vote: positive, body: write_versions, rb_deps)
14: }
15:
16: On receiving a commit request {
17:    in: Transaction T
18:    add_to_history(T)
19:    Pendings.remove(T)
20:    for o_w ∈ T.writeset do
21:        if o_w.version > get_storage_version(o_w) then
22:            apply(o_w)
23:            Readers[o_w.key] = empty list
24:    for o_r ∈ T.readset do
25:        if o_r.version == get_storage_version(o_r) then
26:            Readers[o_r.key].append(T.Id)
27: }
28:
29: On receiving an abort request {
30:    in: Transaction T
31:    Pendings.remove(T)
32: }
33:
34: function detect-conflicts(in: T, Set out: decision){
35:    for transaction T_p ∈ Set do
36:        if T_p.writeset ⋂ (T.readset ⋃ T.writeset) != φ OR
37:            T_p.readset ⋂ T.writeset != φ then
38:                return reject
39:    return validated
40: }
41:
42: function verify-read-versions(in: T, out: decision){
43:    for each o_r in T.readset do
44:        if o_r.version != get_storage_version(o_r) then
45:            return reject
46:    return validated
47: }
48:
49: function get-rb-transactions(in: T, out: rb-txns){
50:    rb-txns: list, initially empty
51:    for each o_w in T.writeset do
52:        rb-txns.add(Ids in Readers[o_w.key])
53:    return rb-txns
54: }
55:
56: function get-write-versions(in: T, out: list){
57:    for each o_w in T.writeset do
58:        write_versions[o_w] = get_storage_version(o_w)
59:    return write_versions
60: }
```

T's read or write-sets and the write-set of pending transactions. Also, the read-set of T is verified by the function verify-read-versions() shown in lines 42-47. A conflict is detected if any read version in T is not equivalent to the version in the current storage. If a conflict is detected by either these two functions, a negative vote is sent back (line 8). Otherwise, dependencies are collected to be sent back with the positive vote (line 10-13). Current version numbers of objects in the write-set are collected as a set called *write_versions* using the function get-write-versions() (lines 56-60). Also, read-before dependencies, *rb_deps*, are collected by the function get-rb-transactions() (lines 49-54). For each object in the write-set, the function returns the corresponding list in *Readers* which contains the read-before transactions Ids. Preparing concludes by sending a positive vote accompanied with *write_versions* and *rb_deps* back to the client (line 13).

Commit requests. Handling commit requests is shown in lines 16-27. A commit request is a command from the client to the replica to incorporate T's write-set. The committed transaction's information is forwarded to RoT-Storage in line 18 and is removed from the *pendings* list. Then, the write-set is applied to the current storage using Thomas write rule [17] (lines 20-23). The rule is to write the data object only if it is a more recent version than what already exists in current storage. When a data object in the write-set is applied, the *Readers* list is updated by emptying the list corresponding to the data object. This is because a new version is written and the previous *Readers* list is valid for the previous version. The final step is to update the *Readers* list to reflect that T reads its read-set (lines 24-26). If the read version is the one that exists in current storage, then T's Id is appended in the readers list of that data object.

Abort requests An abort request for transaction T (lines 29-32) removes it from *Pendings* if it has been added there previously.

RoT-Storage's dependency tracking and read-only transactions

So far, we have presented the protocol for read-modify transactions that changes the state of Ogre's current storage. Only committed transactions are forwarded to the RoT-Storage. Here, we will show the algorithms used to ingest committed transactions in RoT-Storage (Algorithm 8). RoT-Storage supports efficient read-only transactions (Definition 3).

Multi-versioned store. Multi-versioning is a common way to remove contention between read-modify transactions and read-only transactions [95, 96, 97]. The choice of a multi-versioned data structure is an orthogonal problem to Ogre. We employ a simple back-chaining technique where a version of a data object has a pointer to the most recent older version, *i.e.*, a pointer to the previous version. An index points to the most recent version of data objects in RoT-Storage and serves as an access point to all historical data. Each version has a *timestamp*, which represents the time it was written[3]. When applying a transaction's write-set to RoT-Storage, a timestamp is drawn from a monotonically increasing clock that is local to the RoT-Storage. Any transaction T is applied *after* its dependencies. Thus, any transaction T' that is a dependency of T necessarily has a timestamp that is lower than the timestamp of T. Timestamps are meant to preserve dependency information within a single RoT-Storage, hence *no* time synchronization is required. A transaction's timestamp value is used as the timestamp of the transaction's writes.

Dependency tracking and ingestion algorithm. Algorithm 8 shows the method of ingesting committed transactions to RoT-Storage. RoT-Storage maintains a list called $Hist\_Readers$, which has the same function as $Readers$ in current storage, but named differently to easily distinguish them. Also, RoT-Storage has a list called $Buffered$ that con-

[3]Note that the timestamp, used in the multi-versioned RoT-Storage, is different from data object versions, shown previously, which are independent of time and used for conflict detection.

Algorithm 8: Algorithms for dependency tracking and ingestion of transactions at Ogre's RoT-Storage

```
1: STATE
2:     Hist_Readers: A map from data objects to lists of transaction Ids, initially each list is empty
3:     Buffered: A list of transactions, initially empty
4:
5: function add_to_history(in: T){
6:     in: Transaction T
7:     Buffered.append(T)
8:     while ∃_{T_b∈Buffered}(dependencies_satisfied(T_b))
9:         Buffered.remove(T_b)
10:        for o_w ∈ T_b.writeset do
11:            Hist_Readers[o_w.key] = empty list
12:            apply_to_History(o_w)
13:        for o_r ∈ T_b.readset do
14:            Hist_Readers[o_r.key].append(T_b.Id)
15: }
16:
17: function dependencies_satisfied(in: T, out: decision){
18:     rb_deps = get-rb-transactions_H(T)
19:     write_versions = get-write-versions_H(T)
20:     if rb_txns==T.rb_txns AND write_versions==T.write_versions AND verify-read-versions_H(T) is validated then
21:         return True
22:     return False
23: }
24:
25: function verify-read-versions_H(in: T, out: decision){
26:     // The same as verify-read-versions() except that object versions are read from the history storage
27: }
28:
29: function get-rb-transactions_H(in: T, out: rb-txns){
30:     // The same as get-rb-transactions() except that Hist_Readers is used instead of Readers
31: }
32:
33: function get-write-versions_H(in: T, out: list){
34:     // The same as get-write-versions() except that object versions are read from the history storage
35: }
```

tains transactions which are waiting for their dependencies. The function add_to_history() is the entry point to RoT-Storage (lines 5-15). The committed transaction, T, is immediately appended to $Buffered$ (line 7). Then, we look for transactions that can be applied from the $Buffered$ list. A transaction, T_b, is applied if its dependencies are already applied to RoT-Storage (lines 8-14). Dependencies are checked using the function dependencies_satisfied() in lines 17-23. It verifies that the transaction and RoT-Storage have the same read versions, write versions, and read-before dependencies for the objects

Algorithm 9: Ogre's algorithm to execute a read-only transaction at RoT-Storage

```
1: On receiving a read-only transaction{
2: in: A transaction T
3: read_timestamp = get_time()
4: for each operation o in T do
5:     r = read(o)
6:     while r.timestamp > timestamp
7:         r = r.previous
8:     add {o, value} to T.readset
9: Send(to: client, body: T)
10: }
```

accessed by the transaction. Getting these dependencies from RoT-Storage is done using the same functions in current storage (verify-read-versions(), get-rb-transactions(), and get-write-version()). To distinguish these from ones operating on current storage we suffix the functions with "$\_H$" to denote that they are in RoT-Storage (lines 25-35). When a transaction, T_b is found to have satisfied its dependencies, it is first removed from the $Buffered$ list (line 9). Then, $Hist\_Readers$ is updated by adding T_b's id to the lists corresponding to the read objects (lines 10-11), and by clearing the lists corresponding to the written objects (lines 12-13). Note that contrary to the case of updating $Readers$ in current storage, in RoT-Storage we do not check for versions before updating $Hist\_Readers$. This is because they are guaranteed to be equal to what is in T_b. Finally, the write-set is applied to RoT-Storage (line 14).

Executing read-only transactions. Algorithm 9 shows the algorithm to execute a read-only transaction at RoT-Storage. First, a read timestamp is chosen (line 3). Then, for each data object, we read the most recent version (line 5). We then trace back the *previous* pointers until we hit the first version with a timestamp smaller than the read timestamp. This version is added to the read-set (lines 6-8). Finally, the read-set is sent back to the client (line 9).

5.2.3 Summary

We have presented the design of Ogre that extends a majority protocol with dependency tracking and analytics isolation. This entails a design of a commit protocol with dependency collection and an ingestion protocol to maintain a consistent RoT-Storage at all datacenters. The proofs of the correctness of Ogre's commit protocol, the consistency of RoT-Storage snapshots, and that transactions will not be in a deadlock in RoT-Storage are presented next.

5.3 Correctness proofs

In this section, we prove Ogre's correctness (Section 5.3.1) and absence of deadlocks in RoT-Storage (Section 5.3.2).

5.3.1 Correctness proofs

Here, we prove that Ogre guarantees one-copy serializability [12]. We do so by ensuring that any one-copy Serializability Graph (1-SG) resulted from execution history is acyclic, which is a sufficient condition to prove one-copy serializability. The 1-SG consists of nodes, representing transactions, and directed edges, representing relations between transactions. We denote a relation from transaction t_i to t_j with $t_i \rightarrow^x_y t_j$, where y is the type of relation and x is the data object. The types of relations are the following (for transactions t_i and t_j):

- Write-read ($t_i \rightarrow^x_{wr} t_j$): t_j reads a version of x which is written by t_i.
- Write-write ($t_i \rightarrow^x_{ww} t_j$): at any copy, t_j writes a version of x which replaces a version written by t_i. In addition, all transactions which write a common object x must embody a total order. Thus, for any t_i and t_j, writing x, there must be a

directed path of ww edges from t_i to t_j or t_j to t_i. If there are multiple total orders, one can be picked arbitrarily.

- Read-write ($t_i \rightarrow^x_{rw} t_j$): there is a transaction t_k in which there is an edge $t_k \rightarrow^x_{wr} t_i$ and another edge $t_k \rightarrow^x_{ww} t_j$.

In the remainder of this section we will prove that Ogre is correct by showing that the 1-SG of any execution is acyclic. We distinguish between the 1-SG of the execution of read-modify transactions in the current storage, 1-SG$_c$, and the 1-SG of the execution of read-modify and read-only transactions in RoT-Storage, 1-SG$_r$. We begin by the following lemmas:

Lemma 2 *The 1-SG$_c$ embodies a unique total order on writes of each data object x.*

Proof: A transaction t, which writes x, needs a majority vote. All majorities intersect, which means that no other transaction writing x commits concurrently. Transaction t assigns a version number to x, $VN_x(t)$, which is equal to one added to the largest version read in the vote quorum. $VN_x(t)$ is t's index in the total order of write operations on x.

■

Lemma 3 *In current storage, for any conflict relation between two* read-modify *transactions, t and t', from t' to t ($t' \rightarrow^x_y t$), t commits only after t' has already committed.*

Proof: Consider when a transaction t commits, *i.e.*, at the point when a majority of positive votes is received. We will show that each conflict relation from t' to t is such that t' is already committed in a majority:

- $t' \rightarrow^x_{wr} t$: Transaction t only reads an applied write, which necessarily belongs to a committed transaction (Algorithm 7 line 22).

- $t' \rightarrow^{x}_{ww} t$: Lemma 2 shows that there is a unique total order of transactions writing x, thus there is a single ww edge directed to t. Let the version number of x written by t be $VN_x(t)$. This means that there is a ww edge from some transaction t', which wrote a version of x equal to $VN_x(t) - 1$.

  We now show that t' has already committed in a majority when t commits. Transaction t receives the most recent version number of x, $VN_x(t) - 1$, in the replies of the prepare phase. These write versions are collected for committed transactions only (Algorithm 7 line 58). Thus, the transaction, t', which wrote version $VN_x(t) - 1$ has already committed.

- $t' \rightarrow^{x}_{rw} t$: For this conflict relation, there must exist a transaction t'', such that the following conflict relations exist: $t'' \rightarrow^{x}_{wr} t'$ and $t'' \rightarrow^{x}_{ww} t$. This means that t' reads version $VN_x(t'')$, which is equal to $VN_x(t) - 1$ (Lemma 2). t' can only commit after verifying its read-set at a majority. Thus, when t' receives a majority of positive votes: (1) a majority of replicas has version $VN_x(t'')$ as the most recent version, (2) a majority of replicas does not have t in *Pendings*, and (3) a majority of replicas has t' in *Pendings*. t could not have committed when the votes were generated because its information must have been applied or appended to *Pendings* in a majority, which is not the case by (1) and (2). Also, while the votes are sent back for t', t cannot commit because it will be aborted for the conflict with t' in the *Pendings* list, which is ensured by (3). Thus, t cannot have committed until its commit messages are received to at least a majority to clear its information from the *Pendings* list. Thus, t can only commit after t' has committed.

■

Lemma 4 *The 1-SG_c of any execution of read-modify transactions in Ogre is one-copy serializable.*

Proof: This property will be proven via induction. Initially, the 1-SG$_c$ contains a single node which represents a hypothetical transaction which wrote all the initial versions of data objects, call it transaction t_0. Denote 1-SG$_c(t_i)$ to be the state of the graph just when committing a transaction t_i. Thus, initially, the graph state is 1-SG$_c(t_0)$. This is the initial state and has no cycles.

For any 1-SG$_c(t_i)$, there are no edges starting from t_i. This is because all other transactions, t', in 1-SG$_c(t_i)$ are already committed when t_i commits. According to Lemma 3 edges cannot exist from a transaction t_i to transactions which committed earlier to it, t'. What this shows is that when a transaction commits, all new edges are directed to it. Thus, if no cycles existed in the previous graph, no new cycles will emerge. And this is the inductive step. ■

Lemma 5 *The 1-SG$_r$ of any execution of only read-modify transactions in Ogre is one-copy serializable.*

Proof: Committed transactions are forwarded to RoT-Storage. These committed transactions form the acyclic 1-SG$_c$ in the current storage (Lemma 4). Transactions are buffered in RoT-Storage until all their dependencies, which are the three conflict types, are satisfied. Thus, a transaction t_i will be applied only if each transaction t_j, such that there exist $t_j \rightarrow^x_y t_i$, has already been applied. Thus, cycles cannot be formed because all conflict relations take the same direction in the execution history. ■

Lemma 6 *Introducing Ogre's efficient read-only transactions to an acyclic 1-SG$_r$ of Ogre's read-modify transactions in RoT-Storage execution will not introduce any cycles.*

Proof: Call the subgraph of 1-SG$_r$ which contains read-modify transactions only, 1-SG$_{r/rm}$. Lemma 5 shows that 1-SG$_{r/rm}$ is acyclic. Now consider adding read-only transactions to form the whole graph, 1-SG$_r$. The subgraph of 1-SG which contains

read-modify transactions only has exactly the same edges in 1-SG$_{r/rm}$. This is because read-only transactions only contain read operations, and thus, new conflict relations in 1-SG compared to 1-SG$_{rm}$ are from or to read-only transactions. The new conflict relations resulting from introducing read-only transactions to 1-SG$_r$ are either write-read relations to read-only transactions or read-write relations from read-only transactions.

We will show that these new relations do not introduce any cycle. In lemma 5, we showed that all conflict relations in RoT-Storage take one direction in the execution history. This means that for any two transactions, t_i and t_j, there is a conflict relation from t_i to t_j if and only if the timestamp of t_i is smaller than the timestamp of t_j. When a read-only transaction, t, is introduced in the execution history, it is assigned a timestamp, $ts(t)$. The read-only transaction reads from transactions with timestamps lower than $ts(t)$. This means that any t', such that there exist $t' \rightarrow^x_{wr} t$, has a lower timestamp than t, *i.e.*, $ts(t') < ts(t)$.

For each write-read relation, $t' \rightarrow^x_{wr} t$, a read-write relation is introduced from t to t'' which wrote the version of x equal to $VN_x(t) + 1$. Transaction t'' is guaranteed to have a timestamp larger than the timestamp of t. The reason is that t reads the most recent version as of the transaction's read timestamp. Thus, read-write relations introduced by t, $t \rightarrow^x_{rw} t''$, are all such that $ts(t) < ts(t'')$.

Incorporating a read-only transaction in 1-SG$_r$ attains the property of having conflict relations directed to transactions with higher timestamps. Because all relations follow a single direction in execution history, no cycles can be formed. ■

Theorem 2 *Ogre is one-copy serializable.*

Proof: Read-modify transactions operating on current storage are one-copy serializable as shown in Lemma 4. Read-only transactions served from RoT-Storage read a correct snapshot of the history and are one-copy serializable as shown in Lemma 6. ■

5.3.2 Deadlock freedom in RoT-Storage

Transactions are buffered in RoT-Storage waiting for their dependencies. We show here that no deadlocks are possible.

Theorem 3 *Ogre guarantees that no two transactions depend on each other in RoT-Storage's buffer.*

Proof: Each transaction, t, has three types of dependencies. Transaction t only depends on the committed transaction before it commits. We show this for each dependency type:

- Read-set versions: The version numbers in the read-set correspond to the transactions which wrote them. This is the write-read relation. Lemma 3 shows that if a write-read relation exists, such that $t \rightarrow^{x}_{wr} t'$, then t commits before t'.
- Write-versions: Write-versions denote the transactions which wrote the objects in the write-set before the transaction. This is the definition of a write-write relation. Lemma 3 shows that if a write-write relation exists, such that $t \rightarrow^{x}_{ww} t'$, then t commits before t'.
- Read-before dependencies: Read-before transactions are the ones which read the most recent version of a data object in the write-set, which denote the read-write relations. Lemma 3 shows that if a read-write relation exists, such that $t \rightarrow^{x}_{rw} t'$, then t commits before t'.

Because a transaction only depends on a committed transactions, no two transactions can depend on each other, which prevent the existence of cycles in the dependency graph. ■

	V	O	C	I	S
V	-	66 (11)	78 (10)	84 (9)	268 (7)
O	66 (10)	-	19 (1)	175 (7)	210 (4.4)
C	78 (9)	19 (1)	-	175 (7)	182 (6)
I	84 (8)	175 (7)	175 (6)	-	194 (4)
S	268 (6)	210 (4)	182 (6)	194 (4)	-

Table 5.1: RTT latencies between different datacenters in milliseconds and the standard deviation in parentheses

5.4 Evaluation

We have evaluated Ogre's performance on 5 Amazon AWS datacenters and compared its performance to Helios [8], Replicated Commit [54] and an implementation of Two-Phase Commit (2PC) over Paxos (2PC/Paxos) that is inspired from Spanner's commit protocol [25].

Objective. The objective of this study is to understand Ogre's performance for both read-modify and read-only transactions. For read-modify transactions, we focus on two measures of performance: transaction latency and throughput. We compare the performance of Ogre's read-modify transactions with the performance of recent geo-replication proposals. For read-only transactions, we focus on measuring their performance which includes latency, throughput, and staleness.

Setup. Amazon Elastic Compute Cloud (EC2) machines are used. These machines are compute optimized (c4.large) and they have two CPU cores and 3.75GB of memory[4]. Five datacenters were used in the following locations: California (C), Virginia (V), Oregon (O), Ireland (I), and Singapore (S). The average RTT latencies observed are shown in Table 5.1. These RTT numbers are sampled over 24 hours. The standard deviations of the samples are shown in parentheses. Each datacenter hosts a full replica of the data. HBase [88] is used as an underlying data store.

[4]These machines are smaller and less expensive than the machines used in prior studies [8]. Thus, the magnitude of throughput results is smaller than earlier studies.

Workload. Dedicated machines in each datacenter are used to simulate application clients. We use Transactional YCSB (T-YCSB) [90], a multi-record transactional benchmarking framework, for our evaluations. T-YCSB, an extended version of YCSB [89], generates transactional workloads. It issues transactions that consist of a set of read and write operations, where each operation accesses a different record of the data store. Each client can have one outstanding transaction at a time. Clients issue transactions as fast as they can unless mentioned otherwise. An operation is either a read or a write to a key from a pool of 50000 keys. The key is chosen using a Zipfian distribution. The key range is set to be relatively small with a Zipfian access distribution to observe the performance of Ogre under contention. Each read-modify transaction contains five operations. Half of these operations are reads and the other half are writes. Data points shown in this section are sampled over a duration of a single minute.

5.4.1 Systems for comparison

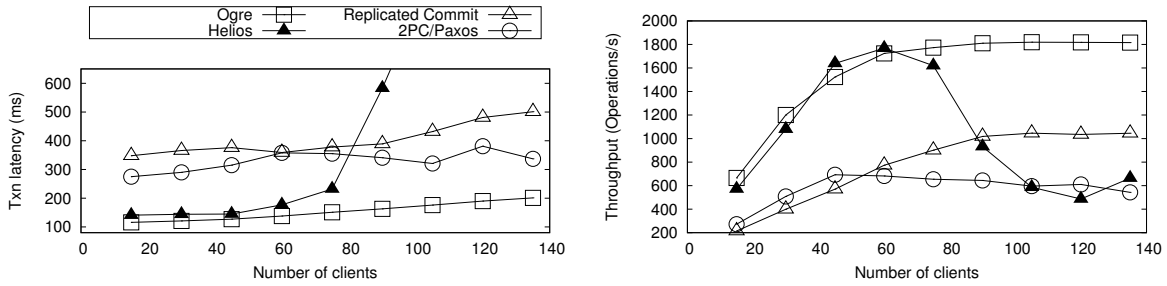
We report Ogre's performance results compared to Helios [8], Replicated Commit [54], and a Two-Phase Commit over Paxos protocol (2PC/Paxos) that is inspired from Spanner [25].

Helios is commit protocol for geo-replicated data based on loose time synchronization. It builds on a theory of lower-bound transaction latency. A target transaction latency is computed and used as part of the protocol. Datacenters exchange causally-ordered logs of timestamped transactions information. A transaction's timestamp is used to commit it. The transaction's host datacenter computes conflict time ranges with other datacenters, based on the transaction's timestamp. Once all these conflict time ranges are learned by the host datacenter, a commit decision is made for the transaction. Helios clients read from the local storage, thus experiencing a low read latency similar to Ogre. Helios has

the nice feature of targeting the lower-bound latency and allowing the flexibility to set the fault-tolerance level. To match Ogre, and other comparison systems, we set Helios' fault-tolerance level to tolerate 2 datacenter outages. In this case, the optimal latency of each datacenter is the time to communicate with the majority, which is the transaction latency of Ogre. Helios' reliance on causally-ordered logs for information propagation creates a bottleneck as maintaining a log's order is inherently centralized.

Replicated Commit uses Paxos for cross-datacenter replication and Two-Phase Locking (2PL) to prevent conflicts. Clients first perform read operations by attempting to lock the read objects in a majority of datacenters. Write operations are buffered. Then, committing the transaction is done by replicating it using Paxos to all datacenters. As the transaction is received by datacenters, locks are acquired for buffered write operations and read locks are validated. If the locks were acquired and validated at a majority of datacenters, then the client commits the transaction. Thus, its commit latency (time duration of the commit protocol after executing operations) should be in the order of a single RTT to the closest majority, which is the commit latency of Ogre as well. However, Replicated Commit's transaction latency is affected by the read strategy; the client spends more time reading before beginning the commit protocol. Ogre, on the other hand, incurs a small read latency, making the transaction latency smaller than what is obtained by Replicated Commit.

2PC/Paxos uses 2PC to avoid conflicts and Paxos for replication. In this evaluation, the datacenter in Virginia (V) is assigned to be the 2PC coordinator. Clients, scattered across all five datacenters issue reads to the coordinator. The coordinator maintains a lock table. When a read is received, a read lock is placed on the corresponding key. Write operations are buffered. When a transaction is ready, a request to commit is sent to the coordinator with information about all operations. Once the coordinator receives the commit request it tries to acquire the write locks and verifies that the read locks are still



(a) Transaction latency as the number of clients increases. (b) Throughput as the number of clients increases.

Figure 5.3: The average latency and throughput of read-modify transactions

held. If successful, the transaction commits. Then, the coordinator replicates the log to a majority of datacenters using Paxos [26].

5.4.2 Read-modify transactions

We begin by showing the performance of read-modify transactions in Figure 5.3. We performed experiments while varying the number of clients from 15 to 135 clients. All Clients issue read-modify transactions back-to-back. The numbers shown are for Ogre compared to the other systems. We desire in this study to demonstrate that although Ogre has a special support for read-only transactions, its read-modify transactions performance is comparable to recent geo-replication proposals.

Transaction latency. Transaction latency is the time required to execute operations and commit the transaction. The transaction latency numbers are shown in Figure 5.3(a). For scenarios with up to 75 clients, the transaction latency of Ogre and Helios are significantly lower than Replicated Commit and 2PC/Paxos. For Example, with 60 clients 2PC/Paxos and Replicated Commit have a transaction latency that is 160% higher than Ogre and 103% higher than Helios. The significant difference in transaction latency is mainly due to the read strategy of Helios and Ogre in which reads are served locally.

Helios performance degrades with more than 75 clients. This thrashing behavior is a signal that the workload is stressing a bottleneck. Helios relies on ordered log propagation to communicate transaction information. This limits the scalability of Helios. Other systems in the experiment experience a gradual increase in transaction latency as the number of clients increases. For example, Ogre's transaction latency difference from the case of 15 clients to the case of 135 clients is 73%. Similarly, the difference of Replicated Commit and 2PC/Paxos are 44% and 22%, respectively.

Throughput. One important aspect of data systems is the ability to achieve high throughput with fewer resources. Here, we measure two metrics: (1) the peak throughput that is achieved by Ogre and the comparison systems, and (2) the amount of resources (application clients) needed to converge to the peak throughput. Figure 5.3(b) shows the throughput results for scenarios with different numbers of clients.

The figure shows the throughput of successfully committed transactions in operations per second. For 15 clients, Ogre achieves a throughput that is 210% higher than Replicated Commit, 144% higher than 2PC/Commit, and just 16% higher than Helios. As the number of clients increases, the throughput of Ogre increases gradually. The increase in throughput is more prominent in the beginning, where the throughput difference between the cases of 30 clients and 15 clients is 535 operations per second, which is only 19.5% lower than the throughput of 15 clients. As the number of clients gets larger the increase in throughput diminishes. Ogre converges to a peak throughput around 1800 operations per second. Converging to the peak throughput is first observed by 75 clients. Helios follows a similar pattern to the peak throughput, which is also around 1800 operations per second observed by 60 clients.

Replicated Commit starts with the lowest throughput with 214 operations per second for 15 clients. As more clients are introduced to the system, Replicated Commit's throughput increases. Replicated Commit converges to a throughput slightly higher than

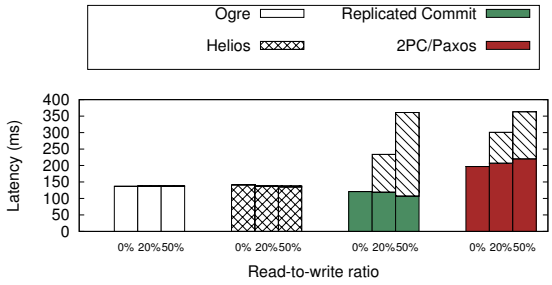


Figure 5.4: Commit and read latency with different read-to-write ratios

1000 operations per second. Like Ogre, Replicated Commit needs 90 clients to achieve the peak throughput. 2PC/Paxos has a different pattern than Ogre and Replicated Commit. 2PC/Paxos throughput starts increasing as more clients are added with 30 and 45 clients. However, after this point, the throughput starts decreasing. The peak throughput with 45 clients is 692 operations per second, which is 154% higher than the throughput with 15 clients.

Converging to a throughput signals a stress bottleneck in the system, where more throughput cannot be achieved. Ogre has a peak throughput that is 80% higher than the peak throughput of Replicated Commit. Ogre and Replicated Commit manage to sustain a throughput that is close to the peak throughput after the convergence point. 2PC/Paxos and Helios, on the other hand, did not sustain the peak throughput; with 135 clients, 2PC/Paxos has a throughput that is 21.5% lower than the peak throughput, and Helios has a throughput that is 62% lower than the peak throughput. The steep decline of Helios' throughput for scenarios with more than 75 clients, matches the steep increase in transaction latency.

5.4.3 Read-to-write ratio

A major overhead of the performance of Replicated Commit and 2PC/Paxos is their read strategies. Now, we will perform experiments with small read-to-write ratios. This

size of read-only transaction (ops)	5	50	500	5000
RM throughput overhead	1%	1%	2%	4%
RM latency overhead	<1%	<1%	<1%	1%
RO throughput (ops/s)	175	1750	17500	97000
RO latency (ms)	23	22	27	75

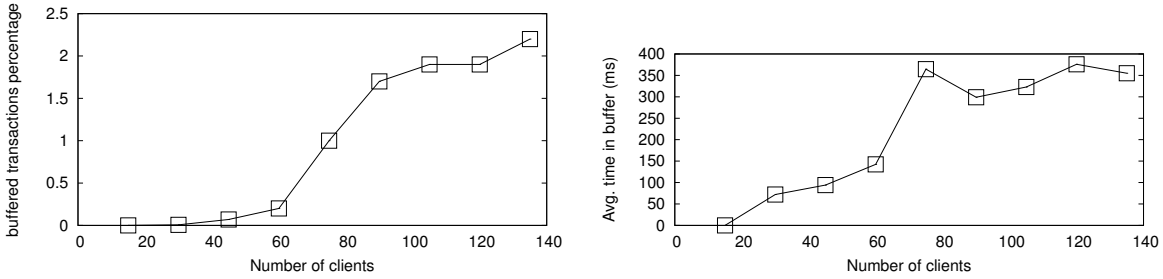
Table 5.2: The performance of Read-modify (RM) and Read-only (RO) transactions on a mixed workload

will show how Ogre and Helios will compare with Replicated Commit and 2PC/Paxos without the overhead of remote reads. Results of this set of experiments are shown in Figure 5.4. In this set, the read-to-write ratio is either 0%, 20% or 50%. The number of issued transaction is limited to 15 transactions per second to isolate the results of this experiment from other performance factors. This will allow us to observe the effect of communication latency alone on latency. In the figure, we show the transaction latency breakdown to commit latency and read latency (the read latency is distinguished by tilted lines on the top portion of bars). Increasing the ratio of reads from 0% to 20% does not have a significant effect on commit latency for all systems. However, it has an effect on read latency. Ogre and Helios has a small increase because they perform reads locally. Replicated Commit and 2PC/Paxos, on the other hand, have an increase of 93% and 53%, respectively. For 50% reads, Replicated Commit and 2PC/Paxos experience another increase compared to the case of 20% reads by 40% and 21%, respectively. 2PC/Paxos's latency increases with a lower magnitude compared to Replicated Commit because clients in the leader (V) are not affected by the increase of read operations, and thus contribute less to the overall average latency increase. Note how the transaction latencies of Ogre, Helios, and Replicated Commit are similar for the case of 0% reads. This demonstrates that the performance advantage of Ogre is mainly due to the read strategy.

5.4.4 Read-only transactions

In this section, we quantify the performance of workloads with both read-only and read-modify transactions. Experiments in Section 5.4.2 show that Ogre's read-modify transactions latency competes with recent geo-replication protocols, although it has the added feature of supporting an efficient execution of read-only transactions. Now, we show that introducing read-only transactions in the workload will yield small overhead and that read-only transactions have a low latency due to their local nature. Also, we will measure the potential staleness observed by read-only transactions.

Performance. Table 5.2 show the results of experiments with mixed workloads. We introduce a read-only client at each datacenter, which issues read-only transactions only. These read-only clients coexist with 60 other clients that issue read-modify transactions. Our aim is to quantify the effect of having read-only clients by comparing to the case of 60 clients with no read-only clients. The read-only clients issue a target number of transactions per second which is 10% of what the 60 clients can achieve individually. We vary this experiment for read-only transactions with different sizes: 5, 50, 500, and 5000 read operations per read-only transaction. The first row shows the effect on read-modify transactions throughput, which is within 4% for all cases. The second row shows the effect on read-modify transactions latency, which is within a single millisecond in all cases. The third row shows the throughput of read-only transactions. For sizes 5, 50, and 500, the target throughput is achieved, which is 10% the number of original transactions times the size of transactions. For read-only transactions with size 5000 operations, the target is not achieved. This is because, for that case, the latency of read-only transactions becomes high enough to restrain the throughput of the available 5 read-only clients. The fourth row shows the latency of read-only transactions, which is between 22ms and 27ms for transactions with sizes 5, 50, and 500. However, the latency becomes 75ms for read-only



(a) Percentage of transactions with unmet dependencies when received by the RoT-Storage

(b) Average amount of wait time in the buffer for transactions with unmet dependencies

Figure 5.5: Measuring staleness in RoT-Storage

transactions with 5000 operations.

Serving read-only transactions locally enables read-only transactions' latency to be low compared to read-modify transactions. And not interfering with read-modify transactions enables read-only transactions to execute without significant overhead to read-modify transactions.

RoT-Storage staleness. Committed transactions are forwarded to RoT-Storage. But, a transaction's information is not applied to RoT-Storage until all its dependencies are met. While waiting for its dependencies to arrive, a transaction t is placed in the RoT-Storage's buffer. A read-only transaction does not see the effect of buffered transactions, and thus potentially reads a stale version.

To measure the staleness caused by the wait for dependencies, we quantify the percentage of transactions that have ever been placed in the RoT-Storage's buffer. For example, if the percentage of buffered transactions is 5%, this means that 95% of transactions are applied immediately to RoT-Storage. The results for scenarios with different numbers of clients are shown in Figure 5.5(a). The percentage of buffered transactions is under 0.5% for scenarios with up to 60 clients. For larger numbers of clients, the percentage of buffered transactions jumps to around 2%. This jump correlates with the convergence to the peak throughput, that we showed in Section 5.4.2.

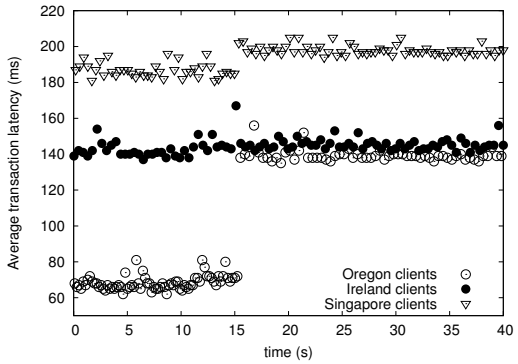


Figure 5.6: A time series of transactions latency of Ogre with an outage of datacenter California at time 15

Figure 5.5(b) shows the average wait time in the RoT-Storage's buffer. Note that this wait time is experienced only by transactions with unmet dependencies (less than 2.5% of transactions in our scenarios). For scenarios with up to 60 clients, the average wait time in the buffer is less than 150ms. For larger numbers of clients, the average wait time jumps to around 350ms. This jump correlates with the increase in the percentage of buffered transactions and with the convergence to the peak throughput.

5.4.5 Fault tolerance

Figure 5.6 shows Ogre's resilience to failures. The figure shows a time series of read-modify transactions latencies. Read-only transactions latencies are not affected as they are local to the datacenter. Each point corresponds to a transaction. Its position in the x-axis is its commit time. The experiment lasts for 40 seconds. At time 15, a failure is induced by killing the Ogre instance at datacenter C. The transaction latency points are samples of transactions at O, I, and S. Clients at O and S are affected by the failure. The latency of transactions jumps from being around 70ms to around 140ms for O, and jumps from being around 180ms to around 200ms for S. This is because clients cannot get votes from C and need to go farther for a majority vote. The increased latencies match

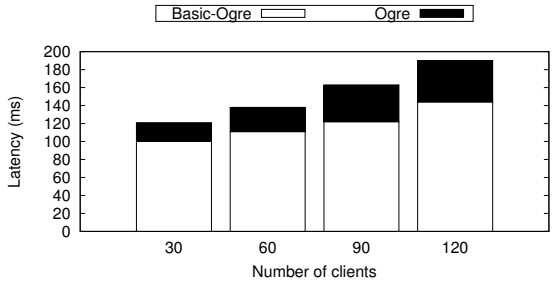


Figure 5.7: The performance cost of tracking dependencies and maintaining RoT-Storage

the RTTs to the closest majority without C (see Table 5.1). Clients at I, however, are not affected by the failure. This is because, for I, the latency to a majority without C is equivalent to the latency to a majority with C. Another effect of C's failure is due to C's pending transactions at other replicas. These transactions cause conflicting transactions to abort for the whole duration of C's outage.

5.4.6 Performance cost of tracking dependencies and maintaining RoT-Storage

Supporting efficient read-only transactions in Ogre incurs an overhead compared to traditional majority protocols. This overhead is due to the computation, communication, and footprints of dependency tracking and maintaining RoT-Storage. Here we quantify Ogre's overhead in comparison to a stripped-down version of Ogre that we will call Basic-Ogre. Basic-Ogre does not perform the actions needed for dependency tracking. Clients do not collect read-before transactions. And Basic-Ogre replicas do not maintain a list of read-before transactions and do not have a coexisting RoT-Storage. Figure 5.7 shows the read-modify transaction latency of four scenarios with different numbers of clients. The overhead of Ogre compared to Basic-Ogre is between 21% and 24% with 30 and 60 clients. This overhead becomes larger for more clients to be between 32% and 34% for 90 and 120 clients.

5.5 Conclusion

Supporting efficient read-only transactions is essential for analytics and maintenance tasks. In this chapter, we propose Ogre, a transaction commit protocol that supports efficient read-only transactions on geo-replicated data. Ogre incorporates *dependency tracking* and *analytics isolation* to overcome some of the limitations of existing read-only transaction protocols. Dependency tracking scales better than existing ordered log shipping protocols for read-only transactions. Also, analytics isolation reduces the overhead of multi-versioning on read-modify transactions.

Chapter 6

COP: Faster Learning from Global-Scale Data by Planning

6.1 Introduction

The increasingly larger sizes of machine learning datasets have motivated the study of scalable parallel and distributed machine learning algorithms [98, 99, 100, 101, 102, 103, 104, 105]. The key to a scalable computation is the efficient management of coordination between processing workers, or workers for short. Some machine learning algorithms require only a small amount of coordination between workers making them easily scalable. However, the vast majority of machine learning algorithms are studied and developed in the serial setting, which makes it arduous to distribute these *serial-based algorithms* while maintaining the algorithm's behavior and goals.

Distributing serial-based algorithms may be performed by encapsulating the algorithm within existing parallel and distributed computation frameworks. These frameworks are oblivious to the actual computation. Thus, the machine learning algorithms may be incorporated as-is without redesign. In this chapter, we consider a framework of

transactions [12, 11] for parallel multi-core execution of machine learning algorithms. A transaction may represent the processing of an iteration of the machine learning algorithm where workers run transactions in parallel. *Serializability* is the correctness criterion for transactions that ensures that the outcome of a parallel computation is equivalent to some serial execution. To guarantee serializability, transactions need to coordinate via consistency schemes such as locking [16] and optimistic concurrency control (OCC) [15].

Recently, coordination-free approaches to parallelizing machine learning algorithms have been proposed [98, 100, 103]. In these approaches, workers do not coordinate with each other thus improving performance significantly compared to methods like locking and OCC. Although these techniques were very successful for many machine learning problems, there is a concern that the coordination-free approach leads to "requiring potentially complex analysis to prove [parallel] algorithm correctness" [99]. When a machine learning algorithm, $\mathbb{A}$, is developed, it is accompanied by mathematical proofs to verify its theoretical properties, such as convergence. These proofs are typically on the serial-based algorithm. A coordination-free parallelization of a proven serial algorithm, denoted $\varphi_{cf}(\mathbb{A})$, is not guaranteed to have the same theoretical properties as the serial algorithm $\mathbb{A}$. This is due to overwrites and inconsistency that makes the outcome of $\varphi_{cf}(\mathbb{A})$ different from $\mathbb{A}$. Thus, guaranteeing the theoretical properties requires a separate mathematical analysis of $\varphi_{cf}(\mathbb{A})$, that although possible [100, 106], can be complex. Additionally, the theoretical analysis might reveal the need for changes to the algorithm to preserve its theoretical guarantees in the parallel setting [100].

Running parallel machine learning algorithms in a serializable, transactional framework bypasses the need for an additional theoretical analysis of the correctness of parallelization. This is because a serializable parallel execution, denoted $\varphi_{SR}(\mathbb{A})$, is equivalent to some serial execution of $\mathbb{A}$, and thus preserves its theoretical properties. We will call parallelizing with serializability, the *universal approach* because serial machine learning algorithms are

applied to it without the need of additional theoretical analysis or changes to the original algorithm.

In this work, we focus on the universal approach of parallelizing machine learning algorithms with serializable transactions. Consistency schemes like locking [107, 16], OCC [100], and others [12] incur a significant performance overhead. Traditional serializability schemes were designed mainly for database workloads. Database workloads are typically arbitrary, unrepeatable units of work that are unknown to the database engine prior to execution. This is not the case for machine learning workloads. Machine learning tasks are well defined. Most machine learning algorithms apply a single iterative task repeatedly to the dataset. Also, the dataset (*i.e.*, the machine learning workload) is typically processed multiple times within the same run of the algorithm, and is potentially used for different runs with different machine learning algorithms. Generally, machine learning datasets are also known, in offline settings, prior to the experiments. These properties of machine learning workloads make it feasible to plan execution. We call these the *dataset knowledge* properties.

We propose Conflict Order Planning (COP) for parallel machine learning algorithms. COP ensures a serializable execution that preserves the theoretical guarantees of the serial machine learning algorithm. It leverages the dataset knowledge properties of machine learning workloads to plan a partial order for concurrent execution that is serializable. It annotates each transaction (*i.e.*, a machine learning iteration) with information about its dependencies according to the planned partial order. At execution time, these *planned dependencies* must be enforced. Enforcing a planned dependency is done by validating that an operation reads or overwrites the correct version according to the plan. This validation is done using a light-weight operation that we call ReadWait. This operation is essentially an arithmetic operation that compares version numbers, which is a much lighter operation compared to locking and other traditional consistency schemes.

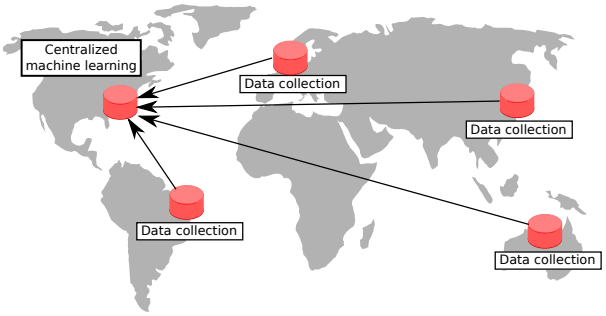


Figure 6.1: The current practice of machine learning of data collected across the world is to batch data at geo-distributed datacenters and send batches to a centralized location that performs the machine learning algorithm

We present background about the problem, the system and transactional machine learning model in Section 6.2. Then, we propose COP in Section 6.3 followed by correctness proofs in Section 6.4. We present our evaluation in Section 6.5. The chapter concludes with a discussion of related work and a conclusion in Sections 6.6 and 6.7.

6.2 Background

In this section, we provide the necessary background for the rest of this chapter. We introduce use cases of planning within machine learning systems in Section 6.2.1. Section 6.2.2 presents the transactional model we will use for machine learning algorithms.

6.2.1 Use Cases

We now demonstrate the opportunity and rewards of planning machine learning execution in three common models of machine learning systems. The first model, Global-Scale machine learning, is the main use case and motivation of this work. However, the solution we propose is general and can be used in various scenarios that we will also present here. We revisit these use cases in the chapter when appropriate to show how COP planning applies to them.

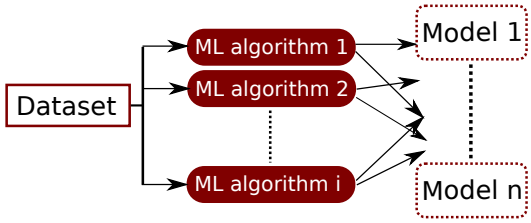


Figure 6.2: A flow diagram of a typical machine learning framework that employs a number of machine learning algorithms to learn models from a dataset

Global-Scale Machine Learning

Online machine learning is the practice of learning from data or events in real time. An example is a web application that collects user interactions with the website and generates a user behavior model using machine learning. Another example is applying machine learning to data collected by Internet of Things (IoT) and mobility devices. Typically, data is *born* around the world, collected at different datacenters, and then sent to a single datacenter that contains a centralized machine learning system. This case is shown in Figure 6.1 where there is a *central datacenter* for machine learning in North America and four other *collection datacenters* that collect and send data. This model has been reported to be the current standard practice of global-scale machine learning [84].

As data is being collected and batched at collection datacenters, there is an opportunity to generate a COP plan. This plan is then applied at the central datacenter for faster execution. A challenge in this model is that data is generated at different locations simultaneously and continuously. In such cases, COP plans for each batch individually at collection datacenters, and then batches are processed at the centralized datacenter in tandem.

Machine Learning Framework

Machine learning and data scientists do not process a dataset only once in their process of analyzing it. Rather, the scientist works on a dataset continuously, experimenting with

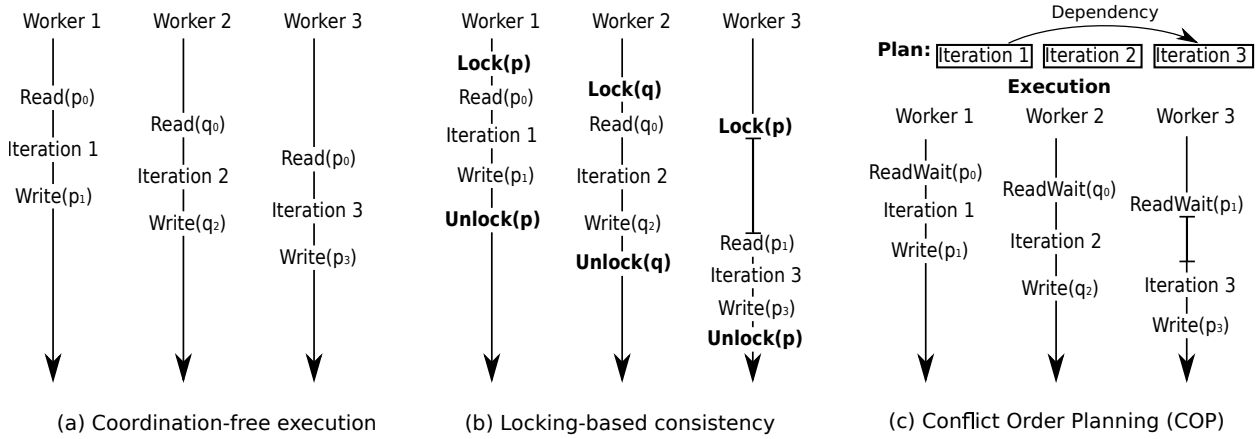


Figure 6.3: Execution of a machine learning algorithm by three workers with different consistency schemes. Each worker processes an iteration of the machine learning algorithm, where the first and third iterations read and update the same model parameter.

different methods and machine learning algorithms to discover what method works best with a dataset. Thus, the same dataset is being processed by many machine learning algorithms repeatedly. Figure 6.2 shows a typical flow diagram of a machine learning framework [105, 108, 109]. Multiple machine learning algorithms are applied to an input dataset to produce models of the dataset. Each machine learning algorithm may be applied multiple times with different configuration and parameters, such as the learning rate.

In this model of a machine learning framework, the dataset is being processed many times, once for each generated model. This is an opportunity for COP to perform a planning stage that is then applied to all runs.

Dataset Loading, Preprocessing and Execution

In addition to the opportunities for planning shown in use cases of machine learning systems, there is an opportunity for planning even in a single execution of a machine learning algorithm on a single dataset. This is because, typically, two tasks are performed

prior to a machine learning algorithm execution: (1) Loading the dataset to main memory. Before execution, the dataset is stored in persistent storage, such as a disk. While loading the dataset from persistent storage, there is an opportunity to perform additional work to plan the execution. Our experiments demonstrate that planning while loading the dataset introduces a small overhead between 3% and 5% (Section 6.5.3).

Datasets are also typically preprocessed for various purposes such as formatting, data cleaning, and normalization [109]. Preprocessing is normally performed on the whole dataset, thus introducing an opportunity to plan execution while preprocessing is performed.

Even in the case of a dataset that is already preprocessed, loaded and ready to be learned, there is another opportunity to plan execution. A machine learning algorithm processes a dataset in multiple *rounds* on the dataset that we call epochs. Thus, planning during the first epoch will be rewarding for the execution of the remaining epochs.

In Section 6.3 we introduce COP planning algorithms and discuss their application to the various use cases we have presented.

6.2.2 Transactional Model of Machine Learning

A machine learning algorithm creates a mathematical model of a problem by iteratively learning from a dataset. The mathematical model of a machine learning algorithm is represented by *model parameters*, P, or parameters for short. For example, the mathematical model of linear regression takes the form $y = \sum_{i=1}^{n} \beta_i x_i + \epsilon$. The model parameters consist of the variables of the model, namely the vector of coefficients, β, and ϵ. The machine learning algorithm uses the dataset to estimate the parameter values that will result in the best fit to predict the dependent variable, y.

A dataset, $\mathbb{D}$, contains a number of samples, where the i^{th} sample is denoted $\mathbb{D}_i$. Each

sample contains information about a subset of the parameters and the dependent variable corresponding to them. To distinguish between model parameters and parameter values in samples, we call the parameter values in samples *features*. For example, a dataset may contain information about movies. Each sample contains a list of the actors in a movie and whether the movie has a high rating. A mathematical model can be constructed to predict whether a movie has a high rating given the list of actors in it. Each parameter in the model corresponds to an actor. A sample contains a vector of feature values, where a feature has a value of 1 if the actor corresponding to it is part of the movie and 0 otherwise. A sample in the dataset also contains whether the movie has a high rating. Using the dataset, the mathematical model is constructed by estimating parameter values. These parameter values can then be utilized in the mathematical model to predict whether a new movie will have a high rating based on the actors in it.

Estimating model parameters is performed by iteratively learning from the dataset. Each iteration processes a single sample or a group of samples to have a better estimate of model parameters. An *epoch* is a collection of iterations that collectively process the whole dataset once. Machine learning algorithms run for many epochs until convergence. For example, Stochastic Gradient Descent (SGD) processes a single sample in each iteration. In an iteration, gradients are computed using a cost function to minimize the error in estimation. The gradients are then used to update the model parameters.

Machine learning algorithms are typically studied and designed for a serial execution where iterations are processed one iteration at a time. A straightforward approach to parallelizing a machine learning computation is to make workers process iterations concurrently, where each worker is responsible for the execution of a different iteration. Executing iterations concurrently may lead to conflicts among some of the updates from different workers, *e.g.*, updates from different workers to the same model parameters may overwrite each other. This means that the behavior of the algorithm no longer resembles

Algorithm 10: Processing an iteration as a transaction

```
1: procedure PROCESS TRANSACTION T_i
2:     μ ← P.read(T_i.read-set)
3:     δ ← ML_computation(μ, T_i.sample, T_i.write-set)
4:     P ← δ
```

Algorithm 11: Parallel machine learning algorithm with Optimistic Concurrency Control

```
1: procedure PROCESS TRANSACTION T_i
2:     μ ← P.read_versioned(T_i.read-set)
3:     δ ← ML_computation(μ, T_i.sample, T_i.write-set)
4:     ATOMIC{
5:         P.validate(μ.versions)
6:         if not validated: abort or restart
7:         P ← δ
8:     }
```

the intended serial execution of the machine learning algorithm.

Figure 6.3(a) illustrates the possibility of data corruption. Three workers are depicted processing three iterations of a machine learning algorithm concurrently. Each iteration reads a subset of the model parameters, computes new estimates of a subset of the model parameters, and finally writes them. In the figure, iterations 1 and 3 read and update the model parameter p and iteration 2 reads and updates the model parameter q. Iterations 1 and 3 read the same version of the parameter p, denoted p_0, and use it to calculate the new parameter value of p. Worker 1 writes the new state of p denoted p_1 and then worker 3 writes the new state of p denoted p_3. In this scenario, the work of worker 1 is overwritten by worker 3. Meanwhile, iteration 2 reads and updates parameter q, which does not corrupt the work of other iterations because it is not reading or writing parameter p.

Serializability can be the correctness criterion for parallel machine learning algorithms [99]. Serializability theory abstracts access to shared data by using the concept of a transaction [11] where a transaction is a collection of read and write operations on data objects. A data object is a subset of the shared state. The computation of an iteration i

of a machine learning algorithm may be abstracted as a transaction, T_i, by considering reads of the model parameters as reads of data objects and writes to the model parameters as writes to data objects. We will denote the collection of model parameters by P, where $P[x]$ is the value of model parameter x. The parameters that are read by a transaction are denoted as $T_i.read\text{-}set$. Similarly, we will denote the parameters that are written by the transaction as $T_i.write\text{-}set$. The sample's data that is processed by iteration i is denoted by $T_i.sample$, where i is the id of the transaction. In the rest of the chapter, we will use the terminology of transactions when appropriate, where a transaction is an iteration, and a data object is a model parameter.

The processing of a transaction follows the template in Algorithm 10 which is a transaction processing template that does not perform any coordination and is only serializable if run sequentially. The transaction template algorithm first reads the model parameters declared in the read-set and cache them locally as μ (line 2). Then, the read parameter values μ and the data sample information, $T_i.sample$, are used to compute new values of the parameters declared in the write-set (line 3). The new values are computed according to the used machine learning algorithms, and they are buffered locally as δ. Finally, the new parameter values, δ, are applied to the shared model parameters, P (line 4).

Serializability guarantees the illusion of a serial execution while being oblivious of the semantic computation performed within the transaction. Thus, it may be applied to machine learning algorithms. Serializability is achieved by ensuring that if some transactions conflict with each other, then they will not be executed concurrently. Detecting conflicts between concurrent transactions requires coordination among workers via different methods. These methods are diverse with different performance characteristics. We now present common transaction execution protocols that have been used in the context of machine learning algorithms, and we generalize them as transactional patterns

that are oblivious to the machine learning algorithm. Readers familiar with transaction processing may skip to Section 6.2.3.

Locking

One of the most common methods for transaction management is *mutual exclusion* also known as lock-based protocols or pessimistic concurrency control [16]. In the rest of the chapter, we will call it *Locking*. Locking is used in many parallel machine learning frameworks to support serializability [110, 111]. In this method, all read or written model parameters are locked during the processing of the transaction. These locks prevent any two transactions from executing concurrently if they access any common objects. Locking may be applied to the transactional pattern of Algorithm 10 by locking all data objects in the read-set and write-set at the beginning. These locks are released only after the transaction updates are applied to the shared model parameters.

Locking prevents conflicts such as the overwrite of worker 3 to worker 1's work in the scenario in Figure 6.3(a). The scenario with Locking is shown in Figure 6.3(b). Workers attempt to acquire a lock on the parameters they read or write before beginning the iteration. Worker 1 acquires the lock for p first and proceeds to compute and update the value of p before releasing the lock. Thus, it prevents worker 3 from overwriting its work because worker 3 will wait until it acquires the lock. Meanwhile, worker 2 acquires the lock for q and process iteration 2 because no other iteration is reading or updating q. This is a serializable execution because it resembles the serial execution of iteration 1, iteration 2, and then iteration 3. However, locking is an expensive operation that leads to a significant performance overhead even for iterations that do not need coordination, such as iteration 2.

Optimistic Concurrency Control

Optimistic concurrency control (OCC) [15] is an alternative to Locking. It performs better for scenarios with low contention, which made it more suitable for machine learning algorithms [99]. However, existing OCC methods for machine learning applications have been only proposed as specialized algorithms for domain-specific machine learning problems. Pan et. al. [99], for example, propose optimistic concurrency control patterns for DP-Means, BP-Means, and online facility location. Unlike these specialized OCC algorithms we present a generalized OCC pattern that can be applied to arbitrary machine learning algorithms.

A general OCC protocol [15] proceeds in three phases shown in Algorithm 11 :

- *Phase I (Execution):* in the execution phase, the transaction's read-set is read from the shared model parameters (line 2). Model parameters in OCC are versioned, where the version number of a parameter is the id of the transaction that wrote it. The read parameter values and the sample information are then used in the machine learning computation (denoted ML_computation) to generate the updates to model parameters, δ (line 3). Note that during this phase no coordination or synchronization is performed.
- *Phase II (Validation):* in this phase we ensure that the read data objects were not overwritten by other transactions during the execution phase (lines 5-6). This is performed by reading the model parameters again after the computation and comparing the read versions to the current versions.
- *Phase III (Commit):* if the validation is successful, the updates, δ, are applied to the global model parameters (line 7).

One requirement for OCC to be serializable is to *perform the validation and commit phases atomically* (lines 4-8) [12]. To perform these two steps atomically, there are two

typical approaches: (1) Execute these steps serially at a coordinator node. This, however, limits scalability, because it means that there is a dedicated worker that is doing the validation and commit for all iterations. Such a method can only be made efficient with domain knowledge about the machine learning problem, which means that the algorithm no longer becomes a general OCC scheme but rather a specialized OCC algorithm [99]. (2) The general approach for validation is to lock the write-set. This is different from Locking in two ways: locks are only held *after* the computation has been performed and only the data objects in the write-set are locked (data objects in the read-set are not locked). Thus, OCC outperforms Locking for cases when the contention is lower, and the write-set is significantly smaller than the read-set. This approach is adopted by recent state-of-the-art OCC transaction protocols in the systems and database systems community [112] and is the method we use in our evaluations.

6.2.3 Performance and Overheads of Consistency Schemes

Consistency schemes, such as Locking and OCC, incur overheads to ensure a serializable execution. These overheads are: (1) *Conflict detection overhead*: this is the overhead due to additional operations needed to detect conflicts, such as locks, atomic sections, and comparing versions. These overheads are incurred even in the absence of a conflict. (2) *Backoff overhead*: this is the wasted time that is incurred due to a detected conflict, such as waiting for a lock to be released, aborting due to deadlock, and failed validation.

For Locking, the conflict detection overhead is due to the operations to acquire and release locks. Even in the absence of conflict, these operations incur an overhead. The backoff overhead for Locking is the time spent waiting for acquired locks to be released. Deadlocks do not occur in our Locking algorithm. This is because locks are acquired in ascending order—locks with lower keys are acquired first. This is possible because the

read and written data objects are declared at the beginning of the execution.

For OCC, the conflict detection overhead is due to the operations to acquire and release locks for the atomic section, and the overhead to validate the read-set. Unlike Locking, OCC locks are only for the data objects in the write-set. The backoff overhead for OCC is due to wasted processing time in the case of an abort and restart when validation fails.

6.3 Conflict Order Planning

In this section, we propose Conflict Order Planning (COP) for parallel machine learning that ensures a serializable execution while reducing the overhead of conflict detection. COP entails no use of locks or atomic blocks, which are expensive operations necessary for existing consistency schemes such as Locking and OCC.

6.3.1 Overview

COP leverages the dataset knowledge property of machine learning workloads: a machine learning algorithm processes a dataset of samples that is known prior to the experiment and is typically processed *multiple* times. This creates the opportunity to plan a partial order of execution to minimize the cost of conflict detection. Dataset knowledge is not manifested in traditional database systems. Thus, existing consistency schemes, such as Locking and OCC are designed with the assumption that they are oblivious of the dataset. The use of traditional database transactional methods leads to a lost opportunity as they do not exploit dataset knowledge. In this section, we propose COP algorithms that exploit dataset knowledge.

The intuition behind COP is to have a planned partial order of transactions prior to execution and then ensure that the partial order is followed during execution. We derive

the planned partial order from an arbitrary starting serial order of transactions. For example, a dataset with n samples will be transformed to n transactions in some planned order $T_1, T_2, \ldots T_n$. We will represent this ordering by the relation $T_i <_o T_j$, where T_i is ordered before T_j. However, during execution, *the order is not enforced between every pair of transactions.* Rather, the order is only enforced for transactions that depend on each other. Thus, if T_2 does not depend on T_1 then a worker may start processing T_2 even if T_1 did not finish. Otherwise, processing T_2 must begin only after T_1 finishes. Thus, the enforced partial order is based on an initial serial order and the conflict relations between transactions.

Definition 4 ***(Planned partial order)*** *There is a planned dependency—or dependency for short—from a transaction T_i to a transaction T_j if the planned order entails T_j reading or overwriting a write made by T_i. We denote this dependency by $T_i \rightsquigarrow_x T_j$ and it exists if all the following conditions are met:*

- *T_i writes the model parameter x ($x \in T_i.write\text{-}set$).*
- *T_j reads or writes the model parameter x ($x \in T_j.read\text{-}set \cup T_j.write\text{-}set$).*
- *T_i is ordered before T_j ($T_i <_o T_j$).*
- *There exists no transaction T_k that is both ordered between T_i and T_j and writes x ($\nexists T_k | x \in T_k.write\text{-}set \wedge T_i <_o T_k <_o T_j$).*

Enforcing the order between transactions that depend on each other is a sufficient condition to guarantee a serializable execution (see Section 6.4.1 for a correctness proof).

COP enforces dependencies by versioning model parameters with the ids of the transactions that wrote them. A transaction only starts execution if the versions it depends on has been written. Consider applying COP to the scenario in Figure 6.3(a). The resulting execution is shown in Figure 6.3(c). Assume that the planned order is to

execute samples 1, 2, and 3, in this order. The partial order consists of a single dependency from iteration 1 to iteration 3, because they both read and write p. Iterations use a special read operation called ReadWait that waits until the version it reads is written by the transaction that it depends on. Iteration 1 is planned to read the initial version of p, denoted p_0, because it is the first ordered iteration to read p. Likewise, iteration 2 is planned to read the initial version of q. Iteration 3 depends on Iteration 1, because they both read and write p. Thus, iteration 3 is planned to read the version of p that is written by iteration 1, denoted p_1. With this plan, workers 1 and 2 process iterations 1 and 2 concurrently after verifying that they have read their planned versions. Worker 3, however, waits until the version p_1 is written by worker 1 and then proceeds to process iteration 3. With COP, workers coordinate without the need of expensive locking primitives. Rather, workers only utilize simple arithmetic operations on the read or written parameter's version number to enforce the plan.

In the remainder of this section, we propose the COP planning algorithm that is used to find and annotate dependency relations between transactions (Section 6.3.2). Then we propose the COP transaction execution algorithm that enforces dependency relations (Section 6.3.3). We discuss the performance benefits of COP in Section 6.3.4.

6.3.2 COP Planning Algorithm

In this section, we present the COP planning algorithm in its basic form—planning prior to execution. Then, we discuss how it can be used to plan in conjunction with the first epoch and how it can be used in cases where there are multiple sources of data.

Algorithm 12: The COP partial order planning algorithm that is performed prior to the experiment.

```
1: Planned_version_list := A list to assign read and write versions initially all zeros
2: version_readers := A list to count the number of transactions that read a version
3: For T_i ∈ Dataset transactions
4:     For r ∈ T_i.read-set
5:         r.planned_version = Planned_version_list[r.param]
6:         version_readers[r.param]++
7:     End For
8:     For w ∈ T_i.write-set
9:         w.p_writer = Planned_version_list[w.param]
10:        Planned_version_list[w.param] = i
11:        w.p_readers = version_readers[w.param]
12:        version_readers[w.param] = 0
13:    End For
14: End For
15: Delete Planned_version_list and version_readers
```

Basic COP Planning

We begin by presenting the basic COP planning strategy. Here, we assume that planning is performed before execution, either in offline settings or while loading the dataset. The objective of the planning algorithm is to annotate the dataset with the planned partial order information. This annotation includes the following:

Definition 5 ***(COP planning and annotation)***

COP planning performs the following two annotations: (1) Read annotation: each read operation is annotated with the version number it should read, and (2) Write annotation: each write operation, w, is annotated with the id of the version it should overwrites, w', and the number of transactions that are planned to read the version w'.

The read annotation's goal is to enforce the order during execution. The write annotation's goal is to ensure that a version is not overwritten until it is read from all the transactions that are planned to read it.

Algorithm 12 shows the steps to annotate transactions with the partial order informa-

tion. The algorithm processes transactions one transaction at a time ordered by some arbitrary order—beginning with T_1 and ending with T_n.

In COP, each read operation in the read-set, r, contains both the read parameter to be read ($r.param$), and the planned read version number ($r.planned\_version$), *i.e.*, the read annotation. A planned version number k means that the transaction must read the value written by transaction T_k. Also, each write in the write-set, w, contains the parameter to be written ($w.param$), the number of transactions that read the previous version ($w.p\_readers$), and the transaction id of the transaction that it is overwriting ($w.p\_writer$), *i.e.*, the write annotation.

The planning algorithm tracks the planned version numbers in a list named $Planned\_version\_list$ as dependencies are being processed. $Planned\_version\_list[x]$ contains the unique transaction id of the most recently planned transaction that writes x. All entries in the list are initialized to 0. Also, the number of version readers are maintained in a list named $version\_readers$. At any point in the planning process, $version\_readers[x]$ contains the number of planned transactions that read the most recently planned written version of x. Both lists are only used within the planning algorithm and are deleted before the execution phase.

The planning of a transaction T_i proceeds by processing the read-set and then the write-set. Each read operation r in the read-set is annotated with a planned version from the $Planned\_version\_list$ (lines 4-5). For example, consider the case where T_i reads model parameter x. Then, there is a read, r, with $r.param$ equals to x. At the time r is being planned, the corresponding value in the list, $Planned\_version\_list[x]$ contains the unique transaction id, k, of the last transaction, T_k, that wrote x. Thus, assigning the planned version of r to k is a way of encoding that the plan is for T_i to read the value of x that was written by T_k. Then, the corresponding number of version readers is incremented (line 6). After processing the read-set, the planning

algorithm processes each write w in the write-set (lines 8-12). Each write is annotated with the previous writer's version number (line 9). Then, the corresponding entry in $Planned\_version\_list$ is updated with the transaction id value i (line 10). Thus, reads of transactions ordered after T_i can observe that they are planned to read T_i's writes. Then, the write is annotated with the number of readers of the previous version (line 11). Finally, the corresponding entry in $version\_readers$ is reset. After all the operations are processed, the lists $Planned\_version\_list$ and $version\_readers$ are deleted.

The outcome of the algorithm is read and write annotations of the whole dataset. The algorithm only requires a single pass on the dataset. In the evaluation section, we perform experiments to quantify the overhead of planning.

Alternative Planning Strategies

The basic COP planning algorithm, presented in the previous section, assumes that planning is performed prior to execution in offline settings or during dataset loading. We now show how to adapt the algorithm to plan in alternative planning scenarios. The first alternative is to plan during the first epoch of the machine learning algorithm's execution. The plan's objective is to annotate transactions with a partial order of a serializable execution. It is possible to execute the first epoch of the machine learning algorithm via a traditional consistency scheme (*e.g.*, Locking) and then annotate the dataset with the partial order of that epoch. Specifically, during the first epoch using Locking, the planning Algorithm 12 is performed for each transaction while all the locks of that transaction are held. Thus, each read is annotated with the read version and each write is annotated with the version it overwrites and the number of readers. After the first epoch, that has passed through the whole dataset, the remaining epochs are processed using COP with the annotated plan. The planning only adds a small overhead to the first epoch, as we discuss in the evaluation section.

Algorithm 13: Parallel execution with COP

```
1: Global num_reads := initially all zeros
2: procedure PROCESS TRANSACTION T_i
3:     For r ∈ T_i.read-set
4:         μ ← P.ReadWait(r)
5:         num_reads[r.param]++
6:     End For
7:     δ ← ML_computation (μ, T_i.sample, T_i.write-set)
8:     For w ∈ δ
9:         w.version = i
10:        While w.p_readers ≠ num_reads[w.param] OR
       w.p_writer ≠ P[w.param].version
11:            Wait
12:        End While
13:    num_reads[w.param] = 0
14:    End For
15:    P ← δ
```

Another alternative is to plan when the dataset is being generated online from multiple sources, in cases such as the global analytics scenario in Section 6.2.1. In such a scenario, planning can be done at each source for batches of samples using Algorithm 12. Then, at the centralized location, the machines learning algorithms process batches in tandem. The dependencies of a batch are transposed to previous batches. For example, consider two batches b_1 and b_2, where b_1 is processed in the centralized location prior to b_2. The transactions in b_2 that have dependencies on the initial version, according to Algorithm 12, are transposed to the most recent version written by b_1. For example, the first transaction that accesses x in b_2 will be annotated as reading the version 0. However, the centralized location will translate this as an annotation to wait for the last version written by b_1.

6.3.3 Planned Execution Algorithm

We present the COP execution algorithm (shown in Algorithm 13) that processes transactions in parallel according to a planned partial order. We associate each model parameter with a version number that corresponds to the transaction that wrote it, *e.g.*,

$P[x].version$ is the current version number of model parameter x. A list of the number of version readers for model parameters, num_reads, is maintained and accessible by all workers. For example, a value for $num\_reads[x]$ of 3 means that so far, three transactions read the current version of x.

Dependencies between transactions are enforced by ensuring that read operations read the planned versions. T_i's read-set is read from the shared model parameters, P (lines 3-5). The ReadWait operation blocks until the annotated planned version is available. The implementation of ReadWait simply reads both the data object and its version number. Then, it compares the version number to the annotated read version number. If they match, the read data object is returned; otherwise, the read is retried until the planned version is read.

After reading the planned version, the number of version readers is incremented (line 5). Then, the transaction execution proceeds by performing the machine learning computation using the read model parameters and the data sample's information (line 7). Writes to the model parameters computed by the machine learning computation, δ, are buffered before they are applied to the model parameters (lines 8-13). First, each write, w, is tagged with a version number equal to the transaction's id (line 9). Thus, future transactions that read the state can infer that T_i is the transaction that wrote these updates. Then, the algorithm waits until the previous version has been read by all planned readers by making sure that the number of version readers is equal to the planned number of readers of that version and by making sure that the current version is identical to $w.p\_writer$ (lines 10-11). Since we are writing a new version, the corresponding entry in num_reads is reset to 0. The writes are then incorporated in the shared state (line 15).

6.3.4 Performance and Overheads

In Section 6.2.3 we discussed two overheads of consistency schemes: conflict detection overhead and backoff overhead. The backoff overhead incurred in COP is similar to Locking and OCC, *i.e.*, transactions wait until conflicting transactions complete. COP's goal is to minimize the other source of overhead: conflict detection overhead that is incurred whether a conflict is detected or not. In COP, the conflict detection overhead is due to: (1) The validation using the ReadWait operation, and (2) Validation that each write operation's previous readers have already read the previous version. These two tasks are performed via arithmetic operations and comparisons only, without the need for expensive synchronization operations like acquiring and releasing locks. This is the main contributor to COP's performance advantage.

6.4 Correctness Proofs

In this section, we present two proofs. The first proves that COP is serializable and the second proves that deadlocks do not occur in COP.

6.4.1 COP Serializability

We prove the correctness of COP and that it ensures a serializable execution that is equivalent to a serial execution. We use a serializability graph (SG) to prove COP's serializability [12]. A protocol is proven serializable if the SGs that represent its possible executions do not have cycles. A SG consists of nodes and edges. Each node represents a committed transaction. A directed edge from one node to another represents a conflict relation. There are three types of conflict relations (edges) in SGs:

- *Write-read (wr) relation*: This relation is denoted as $T_i \rightarrow_{wr} T_j$, which means that there is a *wr* edge from T_i to T_j. This relation exists if T_i writes a version of a data object x and T_j reads that version.

- *Write-write (ww) relation*: This relation is denoted as $T_i \rightarrow_{ww} T_j$, which means that there is a *ww* edge from T_i to T_j. This relation exists if T_i writes a version of a data object x and T_j overwrites that version with a new one.

- *Read-write (rw) relation*: This relation is denoted as $T_i \rightarrow_{rw} T_j$, which means that there is a *rw* edge from T_i to T_j. This relation exists if T_i reads a version of a data object x and T_j overwrites that version with a new one. If this edge exists between two transactions ($T_i \rightarrow_{rw} T_j$) then it must be the case that there exists a transaction T_k that writes x with the following conflict relations: (1) A write-write conflict relation from T_k to T_j ($T_k \rightarrow_{ww} T_j$), and (2) a write-read conflict relation from T_k to T_i ($T_k \rightarrow_{wr} T_i$).

Lemma 7 *For any conflict relation $T_i \rightarrow T_j$ in SG of a COP execution, the following is true: $T_i <_o T_j$, where $<_o$ is the ordering relation of the initial planned order.*

Proof: Assume that the data object that causes the conflict relation is data object x. We prove this lemma for the three conflict relations:

- *Write-read (wr) conflict relations ($T_i \rightarrow_{wr} T_j$):* according to Definition 4 a transaction T_j is planned to read from a transaction T_i if there is an ordering dependency $T_i \rightsquigarrow_x T_j$. One of the conditions of this ordering dependency is that T_i is ordered before T_j ($T_i <_o T_j$). In the implementation algorithm, this is enforced by the ReadWait operation (see Algorithm 13 lines 3-4).

- *Write-write (ww) conflict relations ($T_i \rightarrow_{ww} T_j$)):* according to Definition 4 a transaction T_j is planned to overwrite a value written by transaction T_i if there is an

ordering dependency $T_i \rightsquigarrow_x T_j$. One of the conditions of this ordering dependency is that T_i is ordered before T_j ($T_i <_o T_j$). In the implementation algorithm, this is enforced by the check of $w.p\_writer$ (see Algorithm 13 lines 10-11).

- *Read-write (rw) conflict relations ($T_i \rightarrow_{rw} T_j$):* this relation implies the existence of a transaction T_k with the relations $T_k \rightarrow_{wr} T_i$ and $T_k \rightarrow_{ww} T_j$. According to our analysis in the previous two points, the following is true:

$$T_k <_o T_i \quad and \quad T_k <_o T_j \tag{6.1}$$

Thus, the following ordering dependencies exist:

$$T_k \rightsquigarrow_x T_i \quad and \quad T_k \rightsquigarrow_x T_j \tag{6.2}$$

We now show by contradiction that the following is true: $T_i <_o T_j$. Assume to the contrary that $T_j <_o T_i$ is true. If $T_j <_o T_i$ then according to Equation 6.1 the following is true:

$$T_k <_o T_j <_o T_i \tag{6.3}$$

However, this equation violates one of the definitions in Definition 4 that states that the ordering relation $T_k \rightsquigarrow_x T_i$ that exists according to Equation 6.2 implies that there exists no transaction that is ordered between them and writes x. However, according to Equation 6.3, T_j is ordered between T_k and T_i and it writes x. This violation leads to a contradiction to $T_j <_o T_i$ thus proving that $T_i <_o T_j$. In the implementation algorithm, this is enforced by the check of $w.p\_readers$ (see Algorithm 13 lines 10-11).

The condition of the lemma is proven for all three conflict relations. ■

	Properties				Performance (M transactions/s)			
Dataset	# features	training set size	test set size	avg. txn size	Ideal	COP	Locking	OCC
KDDA [113]	20,216,830	8,407,752	510,302	36.3	7.2	4.1	0.75	0.82
KDDB [113]	29,890,095	19,264,097	748,401	29.4	8.0	5.8	0.95	1.0
IMDB	685,569	167,773		14.6	15.2	11.0	6.7	4.9

Table 6.1: Performance comparison across of COP, Locking, OCC, and Ideal (without conflict detection) for three datasets

Theorem 4 *Conflict Order Planning (COP) algorithms guarantee serializability.*

Proof: According to Lemma 7, a conflict relation $T_i \rightarrow T_j$ in SG means that $T_i <_o T_j$. We need to show that a cycle $T_i \rightarrow \ldots \rightarrow T_i$ cannot exist. Assume to the contrary that such a cycle exists. This means according to Lemma 7 that $T_i <_o \ldots <_o T_i$. Since the ordering relation $<_o$ is transitive this leads to $T_i <_o T_i$, which is a contradiction, thus proving that no cycles exist in the SG of COP executions. The absence of cycles in SG is a sufficient condition to prove serializability [12]. ■

6.4.2 COP Deadlock Freedom

The COP execution algorithm 13 can block in two locations: (1) a read waits for its planned version to be available, and (2) a write waits until all reads of the previous versions and the write of the previous version are complete. In this section we prove that these waits do not cause a deadlock scenario where a group of transactions are waiting for each other. We prove this by constructing a deadlock graph (DG). Nodes in DG are transactions. A directed edge from one transaction to another, $T_i \rightarrow_d T_j$, denotes that T_j may block waiting for a read or a write of T_i.

Lemma 8 *For any edge in DG, $T_i \rightarrow_d T_j$, the following is true: $T_i <_o T_j$, where $<_o$ is the ordering relation of the planned order.*

Proof: An edge $T_i \rightarrow_d T_j$ exists in three cases: (1) T_j reads a version written by T_i, (2) T_j overwrites a version written by T_i, or (3) T_j overwrites a version to be read by T_i. All cases are true in the COP algorithms only if the ordering dependency $T_i \rightsquigarrow T_j$ exists. According to Definition 4, an ordering dependency $T_i \rightsquigarrow T_j$ is only true if $T_i <_o T_j$. ■

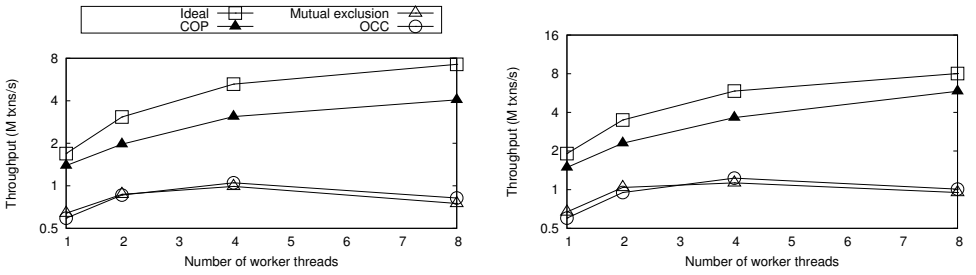
Theorem 5 *Conflict Order Planning (COP) algorithms guarantee deadlock freedom.*

Proof: According to Lemma 8, a dependency relation $T_i \rightarrow_d T_j$ in DG means that $T_i <_o T_j$. We need to show that a cycle $T_i \rightarrow_d \ldots \rightarrow_d T_i$ cannot exist. Assume to the contrary that such a cycle exists. This means according to Lemma 8 that $T_i <_o \ldots <_o T_i$. Since the ordering relation $<_o$ is transitive this leads to $T_i <_o T_i$, which is a contradiction, thus proving that no cycles exist in the DG of COP executions. The absence of cycles in DG is a sufficient condition to prove deadlock freedom. ■

6.5 Evaluation

In this section, we evaluate COP in comparison to Locking and OCC. We also compare with an upper-bound of performance, which is the performance without any conflict detection. We will call this upper-bound the *ideal* baseline, or Ideal for short. Ideal does not guarantee a serializable execution, unlike COP, Locking, and OCC. Thus, Ideal does not guarantee preserving the theoretical properties of the machine learning algorithm.

The transactional framework of machine learning can be applied to a wide-range of machine learning algorithms. For this evaluation, we run our experiments with a Stochastic Gradient Descent (SGD) algorithm to learn a Support Vector Machine (SVM) model. The goal of the machine learning algorithm is to minimize a cost function f. We use a separable cost function for SVM [100]. Each iteration in SGD processes a single sample from the dataset. Gradients are computed according to the cost function. The



(a) Throughput with the KDDA dataset (b) Throughput with the KDDB dataset

(c) Throughput with the IMDB dataset

Figure 6.4: The throughput of Ideal, COP, Locking, and OCC while varying the number of threads for three datasets (log scale is used)

gradients are then used to compute the new values of the model that are relevant to the sample. We apply this machine learning algorithm to the transactional template we presented in Algorithm 10. Each transaction corresponds to an iteration of SGD. The iteration computation (*i.e.*, ML_computation() in the algorithm) represents the gradient computation using the cost function. For this machine learning algorithm, the read and write-sets of a transaction are the features in the corresponding sample, *i.e.*, the features with a non-zero value. In all experiments, we initialize the SGD step size value to 0.1. The step size value diminishes by a factor 0.9 at the end of each epoch over the training dataset. All experiments are run for 20 epochs, where an epoch is a complete pass on the whole dataset.

We implemented COP, Locking, and OCC as a layer on top of the parallel machine learning framework of Hogwild! [100] that is available publicly[1]. The source code is written

[1]http://i.stanford.edu/hazy/victor/Hogwild/

in C++. We use an Amazon AWS EC2 virtual machine to run our experiments. The virtual machine type is c4.4xlarge with 30 GB memory and 16 vCPUs that are equivalent to 8 physical cores (Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz) and 16 hyper threads. Unless mentioned otherwise, the number of worker threads used in the experiments is 8. Our experiments with more than 8 threads show no significant performance difference.

We use three datasets to conduct our experiments, summarized in Table 6.1. The first two datasets are KDDA and KDDB datasets, which were part of the 2010 KDD Cup [113]. KDDA (labeled algebra_2008_2009) has 20,216,830 features and contains 8,407,752 samples in the training set and 510,302 samples in the test set. KDDB (labeled bridge_to_algebra_2008_2009) has 29,890,095 features and contains 19,264,097 samples in the training set and 748,401 samples in the test set. The third dataset is the IMDB dataset[2] that has 685,569 features and contains 167,773 samples. The IMDB dataset is not divided into training and test sets. The average sample (transaction) size of each dataset, which is the number of model parameters represented in each sample, is 36.3 for KDDA, 29.4 for KDDB, and 14.6 for IMDB. In addition to these three datasets, we use synthetic datasets for experiments that require controlling the dataset properties, such as contention.

6.5.1 Throughput

The metric that we are interested in the most is throughput. We measure throughput as the number of processed samples (*i.e.*, transactions) per second. Table 6.1 shows a summary of throughput numbers for the different evaluated methods on the three datasets. COP outperforms Locking and OCC by a factor of 5-6x for KDDA and KDDB. For IMDB, COP's throughput is 64% higher than Locking and 124% higher than OCC. The magnitude of performance improvement of COP compared to Locking and OCC is

[2]http://komarix.org/ac/ds/

influenced by the level of contention in the dataset, *i.e.*, the likelihood of conflict between transactions. Our inspection of the datasets revealed that there is more opportunity for conflict in the KDDA and KDDB datasets than the IMDB dataset. We do not present the statistical properties of the datasets to show this due to the lack of space. However, we perform more experiments in Section 6.5.2 to study the effect of contention on performance. The comparison with Ideal shows that COP's throughput is 27-44% lower than Ideal. This percentage represents COP's overhead to preserve consistency. Although conflicts are planned in COP, there is still an overhead for conflict detection and backoff.

The throughputs of Locking and OCC are relatively close to each other. For KDDA and KDDB, the throughputs of Locking and OCC are within 10% of each other. For IMDB, Locking outperforms OCC by 36.7%. In the case of KDDA and KDDB, the locking contention for both Locking and OCC (to implement atomic validation) dominates performance. In general, OCC benefits in cases where the read-set is larger than the write-set. Because our machine learning workload has a read and write-sets of equal sizes, the advantage of OCC is not manifested (see Section 6.2.3). In IMDB, which is the workload with less contention, Locking outperforms OCC. This is due to the additional work needed to validate transactions by OCC. For the conflict detection overhead, OCC experiences both the overheads of locking and validation, while Locking only experiences the overhead of locking. The overhead of validation is exposed with workloads with less contention because in these cases, locking contention does not dominate performance, *i.e.*, in the case of the KDDA and KDDB datasets, the overhead due to locking contention dominates the validation overhead. We revisit the effect of contention in Section 6.5.2.

In Figure 6.4, we show the performance of the different schemes while varying the number of threads. Increasing the number of threads increases contention. Also, using more cores in the experiment exposes the effect of the underlying cache and cache coherence on the performance of the different schemes. Figure 6.4(a) shows the performance for the

KDDA dataset. Consider the throughput of all schemes with a single worker thread. In this case, there is no conflict or cache coherence overhead. What is observed is the *conflict detection* overhead in isolation (Section 6.2.3). Ideal is only 21% higher than COP in the case of a single worker thread. This shows that the overhead of conflict detection is small compared to Locking and OCC; the throughput of Ideal is 163% higher than Locking and 186% higher than OCC.

For scenarios with more than one worker thread in Figure 6.4(a), the backoff and cache coherence overheads are experienced in addition to the conflict detection overhead. Ideal does not suffer from the backoff overhead because conflicts are not prevented. Also, Ideal has an advantage with the cache coherence overhead compared to the consistent schemes. Unlike COP, Locking, and OCC, Ideal does not maintain additional locking or versioning data that may be invalidated by cache coherence protocols. These factors cause the performance gap between Ideal and the other schemes to grow as the number of threads is increased. COP's throughput, for example, is 17% lower than Ideal with one worker thread, but it is lower by 43% in the case of 8 threads. The contention between cores due to cache coherence limits scalability. Ideal with 8 threads achieves 4 times the performance of the case with a single thread—rather than 8 times the performance in the case of linear scalability. COP with 8 threads achieves 3 times the performance of the case with a single thread. For Locking and OCC, the contention is so severe that performance slightly decreases beyond 4 threads.

We show the same set of experiments for KDDB and IMDB in Figures 6.4(b) and 6.4(c). The experiments with the KDDB dataset show similar behavior to the experiments with the KDDA dataset. One difference is that COP scales better, as the KDDB dataset is sparser than KDDA; for KDDB, COP's throughput with 8 threads is 4 times the throughput with a single thread, rather than a 3x factor with the KDDA dataset. For the IMDB dataset, there is less contention compared to KDDA and KDDB. All schemes—

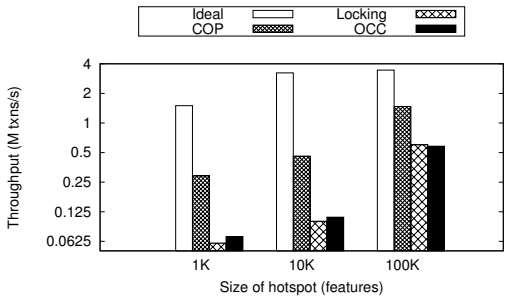


Figure 6.5: Quantifying the effect of contention on performance by experiments on synthetic datasets with varying contention levels

including Locking and OCC—scale with a factor around 4x when increasing the number of threads from 1 to 8. Also, the smaller transaction sizes with the IMDB dataset makes the absolute throughput numbers higher than those with the KDDA and KDDB datasets.

6.5.2 Contention Effect

Contention affects performance because it increases the rate of conflict. A conflict between two transactions causes at least one of them to either wait or restart, thus wasting resources. Here, we quantify the effect of contention on the performance of our consistency schemes. We generate synthetic datasets to give us more flexibility in controlling the contention. The synthetic datasets we generate contain one million samples each. We fix the size of each sample to 100 features, which means that each transaction contains 100 data objects in the read and write-sets. To control the contention, we restrict transactions to a hot spot in the parameter space. Each data object is sampled uniformly from the hot spot. We control contention by varying the size of the hot spot.

Figure 6.5 shows the performance with hot spot sizes of 1K, 10K, and 100K features. Contention leads to a higher conflict overhead and lower performance. This is why consistency schemes perform lower in the highest contention case (1K features) when compared to cases with less contention. The performance improvement factor of the case

with 100K features compared to the case with 1K features is 4x for COP, 8.8x for Locking, and 7.3x for OCC. Ideal also performs lower as contention increases, although it does not face an overhead due to conflicts. The performance of Ideal with 100K features is 131% higher than the performance with 1K features. The reason is that more contention also means more contention on cache lines, leading to a larger overhead for cache coherence.

As contention decreases, the performance gap between the consistency schemes and Ideal decreases. Part of the performance benefit of Ideal compared to the consistency schemes is that Ideal does not block or restart transactions due to conflicts. As contention decreases, this performance benefit diminishes, and the performance of the consistency schemes becomes closer to Ideal. For example, in the high contention case (1K features) Ideal's throughput is 4x the throughput of COP. For the low contention case (100K features), this gap decreases with Ideal's throughput only 34% higher than COP. This is also true for Locking and OCC, where Ideal's throughput is higher than them by a factor of 20-23x in the high contention case, but this factor decreases to around 5x for the low contention case.

Like the performance difference between Ideal and the other consistency schemes, the performance gap between COP and the other consistency schemes (Locking and OCC) also decreases as contention decreases. COP's light-weight conflict detection makes it less prone to conflicts than Locking and OCC because the latency of the transaction is lower. Thus, Locking and OCC suffer from contention more than COP. In the low contention case, COP's throughput is 3.7x higher than Locking and 3.1x higher than OCC. This performance gap decreases in the low contention case where COP outperforms Locking by 46% and OCC by 51%.

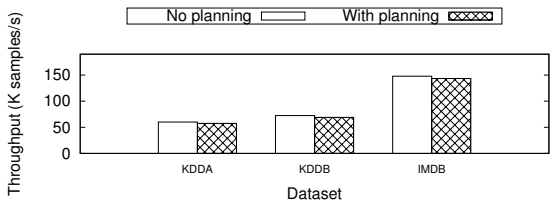


Figure 6.6: A comparison of the loading time of the dataset to main memory with and without order planning.

6.5.3 Planning Overhead

COP's performance advantage is due to having conflicts planned ahead of time. We have outlined in Section 6.2.1 examples of machine learning environments. In these environments, planning can be done in advance, and thus the planning overhead is not observed when the machine learning algorithm is processed using COP. This includes the case of the machine learning framework where a dataset is reused in different experiments and is possibly stored with the annotated plan for future sessions. However, there are cases where machine learning algorithms are used for fresh and raw datasets. In these cases, the planning overhead becomes important.

We performed several experiments to quantify the overhead of planning. We propose two alternatives to plan for a dataset. The first planning strategy is to plan while loading the dataset. A dataset is typically stored in a persistent storage such as a disk. Planning can be done in conjunction with reading the raw dataset from persistent storage and loading it into the appropriate data structures in main memory. Figure 6.6 shows the loading throughput with and without planning for three datasets. Planning only adds a small overhead to loading that we measure to be between 3% and 5%.

The second planning strategy is to plan during the first epoch and then use the plan for later epochs (Section 6.3.2). In the first epoch, a consistency scheme must be used. We run the first epoch using Locking and the rest of the epochs using COP. The throughput of the first epoch is within 1% of the throughput of Locking for all our datasets. The

throughput of the remaining epoch is also within 1% of the performance of COP with offline planning.

6.6 Related Work

The use of transactional and consistency concepts have been explored recently for parallel and distribute machine learning by Pan et.al [98, 99, 101, 104, 114]. Some of these works build consistent algorithms that follow the OCC pattern for distributed unsupervised learning [99], correlation clustering [98, 104], and submodular maximization [101]. These proposals show that domain-specific implementations of OCC—rather than general OCC that we presented in this work—achieve performance close to their coordination-free counterparts while guaranteeing serializability [98, 101].

The study of consistent machine learning algorithms has been motivated by the complexity of developing mathematical guarantees and coordination-free algorithms that are parallel [98, 99, 101, 104, 114]. However, many coordination-free machine learning algorithms were developed [100, 115, 116, 106]. Hogwild! [100], for example, is an asynchronous parallel SGD algorithm with proven convergence guarantees for several classes of machine learning algorithms.

Bounded staleness has been proposed as an alternative to both serializability and coordination-free execution for parallel machine learning. Bounded staleness is a correctness guarantee of the freshness of read data objects. It has been demonstrated for distributed machine learning tasks [117, 118]. Bounded staleness, however, may still lead to data corruption which requires a careful design of machine learning algorithms that leverage bounded staleness.

The concept of planning execution to improve the performance of distributed and parallel transaction processing has been explored in different contexts. Calvin [94] is a

deterministic transaction execution protocol. Sequencing workers intercept transactions and put them in a global order that is enforced by scheduling workers. Calvin is built for typical database transactional workload and thus does not leverage the dataset knowledge property of machine learning workloads. This makes its design incur unnecessary overheads compared to COP for machine learning workloads, such as always having the sequencing and scheduling workers in the path of execution. Schism [119] is a workload-driven replication and partitioning approach. The access patterns are learned from the coming workload to create partitioning strategies that minimize conflict between partitions and thus improve performance. Cyclades [114] adopts a similar approach to Schism for parallel machine learning workloads. Cyclades improves the performance of both conflict-free and consistent machine learning algorithms by partitioning access for batches of the dataset to minimize conflict between partitions. Each partition is then processed by a dedicated thread, leading to better performance. Partitioning for performance complements COP's objective. Whereas partitioning aims to minimize conflict between workers, COP ensures that conflicts are handled more efficiently.

6.7 Conclusion

In this chapter, we propose Conflict Order Planning (COP) for consistent parallel machine learning. COP leverages dataset knowledge to plan a partial order of concurrent execution. Planning enables COP to execute with light-weight synchronization operations and outperform existing consistency schemes such as Locking and OCC while maintaining serializability for machine learning workloads. Our evaluations validate the efficiency of COP on a SGD algorithm for SVMs.

Part III

Global-Scale Communication Infrastructure for Transactions

Chapter 7

Chariots: Global-Scale Log Shipping and Sharing

7.1 Introduction

The explosive growth of web applications and the need to support millions of users make the process of developing web applications difficult. These applications need to support this increasing demand and in the same time they need to satisfy many requirements. Fault-tolerance, availability, and a low response time are some of these requirements. It is overwhelming for the developer to be responsible for ensuring all these requirements while scaling the application to millions of users. The process is error-prone and wastes a lot of efforts by reinventing the wheel for every application.

The cloud model of computing encourages the existence of unified platforms to provide an infrastructure that guarantees the properties needed by applications. Nowadays, such an infrastructure for compute and storage services is commonplace. For example, an application can request a key-value store service in the cloud. The store exposes an interface to the client and hides all the complexities required for its scalability and

fault-tolerance. We envision that a variety of programming platforms will coexist in the cloud for the developer to utilize. A developer can use multiple platforms simultaneously according to the application's needs. A micro-blogging application for example might use a key-value store platform to persist the blogs and in the same time use a distributed processing platform to analyze the stream of blogs. The shared log, as we argue next, is an essential cloud platform in the developer's arsenal.

Manipulation of shared state by distributed applications is an error-prone process. It has been identified that using immutable state rather than directly modifying shared data can help alleviate some of the problems of distributed programming [120, 121, 122]. The *shared log* offers a way to share immutable state and accumulate changes to data, making it a suitable framework for cloud application development. Additionally, the shared log abstraction is familiar to developers. A simple interface of append and read operations can be utilized to build complex solutions. These characteristics allow the development of a wide-range of applications. Solutions that provide transactions, analytics, and stream processing can be easily built over a shared log. Implementing these tasks on a shared log makes reasoning about their correctness and behavior easier and rid the developer from thinking about scalability and fault-tolerance. Also, the log provides a trace of all application events providing a natural framework for tasks like debugging, auditing, checkpointing, and time travel. This inspired a lot of work in the literature to utilize shared logs for building systems such as transaction managers, geo-replicated key-value stores, and others [33, 37, 92, 7, 123, 93, 8].

Although appealing as a platform for diverse programming applications, shared log systems suffer from a single-point of contention problem. Assigning a log position to a record in the shared log must satisfy the uniqueness and order of each log position and consequent records should have no gaps in between. Many shared log solutions tackle this problem and try to increase the append throughput of the log by minimizing the amount

of work done to append a record. The most notable work in this area is the CORFU protocol [124] built on flash chips that is used by Tango [125]. The CORFU protocol is driven by the clients and uses a *centralized sequencer* that assigns offsets to clients to be filled later. This takes the sequencer out of the data path and allows the append throughput to be more than a single machine's I/O bandwidth. However, it is still limited by the bandwidth of the sequencer. This bandwidth is suitable for small clusters but cannot be used to handle larger demands encountered by large-scale web applications.

We propose **FLStore**, a distributed deterministic shared log system that scales beyond the limitations of a single machine. FLStore consists of a group of *log maintainers* that mutually handle exclusive ranges of the log. Disjoint ranges of the log are handled independently by different log maintainers. FLStore ensures that all these tasks are independent by using a deterministic approach that assigns log positions to records as they are received by log maintainers. It removes the need for a centralized sequencer by avoiding log position pre-assignment. Rather, FLStore adopts a *post-assignment* approach where records are assigned log positions *after* they are received by the Log maintainers. FLStore handles the challenges that arise from this scheme. The first challenge is the existence of gaps in the log that occur when a log maintainer has advanced farther compared to other log maintainers. Another challenge is maintaining explicit order dependencies requested by the application developer.

Cloud applications are increasingly employing geo-replication to achieve higher availability and fault-tolerance. Records are generated at multiple datacenters and are potentially processed at multiple locations. This is true for applications that operate on shared data and need communication to other datacenters to make decisions. In addition, some applications process streams coming from different locations. An example is Google's Photon [70] which joins streams of clicks from different datacenters. Performing analytics also requires access to the data generated at multiple datacenters. Geo-replication poses

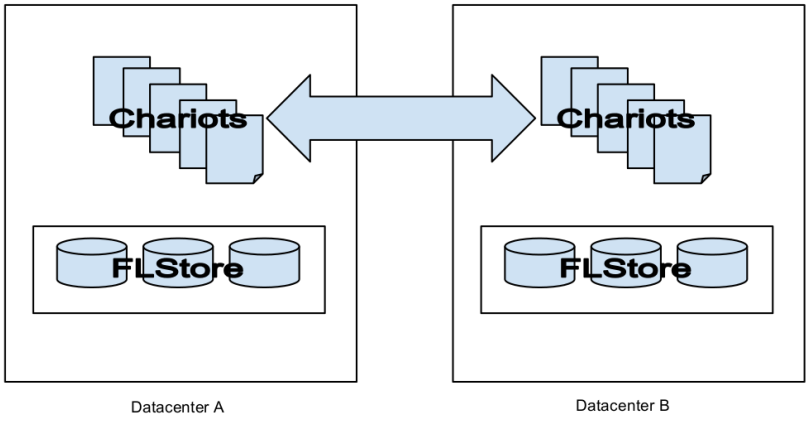


Figure 7.1: The architecture of the proposed systems in this chapter. FLStore is used to manage distributed logs within datacenters and Chariots is used to manage the geo-replication of log entries across datacenters.

additional challenges such as maintaining *exactly-once* semantics (ensure that an event is not processed more than once), automatic datacenter-level fault-tolerance, and handling un-ordered streams.

Chariots supports multiple datacenters by providing a global replicated shared log that contains all the records generated by all datacenters. The order of records in the log must be consistent. The ideal case is to have an identical order at all datacenters. However, it is shown by the CAP theorem [126, 127] that such a **consistency** guarantee *cannot* be achieved if we are to preserve **availability** and **partition-tolerance**. In this work we favor availability and partition-tolerance, as did many other works in different contexts [42, 38, 39, 128]. Here, we relax the guarantees on the order of records in the log. In relaxing the consistency guarantee, we seek the strongest guarantee that will allow us to preserve availability and partition-tolerance. We find, as other systems have [42, 129, 130, 131, 43], that causality [41] is a sufficiently strong guarantee fitting our criterion [132].

In this chapter, we propose a cloud platform that exposes a shared log to applications. This shared log is replicated to multiple datacenters for availability and fault tolerance.

FLStore and Chariots are the two main components of the platform, where FLStore handles distributed logs within the datacenter, and Chariots handles the geo-replication of log entries across datacenters (see Figure 7.1). The objective of the platform's design is to achieve high performance and scalability by allowing seamless elasticity. Challenges in building such a platform are tackled, including handling component and whole datacenter failures, garbage collection, and gaps in the logs. We motivate the log as a framework for building cloud applications by designing three applications on top of the shared log platform. These applications are: (1) a key-value store that preserves causality across datacenters, (2) a stream processing applications that handles streams coming from multiple datacenters, and (3) a replicated data store that provides strongly consistent transactions [7].

The contributions of the chapter are the following:

- A design of a scalable distributed *log storage*, FLStore, that overcomes the bottleneck of a single machine. This is done by adopting a post-assignment approach to assigning log positions.

- Chariots tackles the problem of scaling causally-ordered geo-replicated shared logs by incorporating a distributed log storage solution for each replica. An elastic design is built to allow scaling to datacenter-scale computation. This is the first work we are aware of that tackles the problem of *scaling geo-replicated shared logs through partitioning*.

The chapter proceeds as the following. We first present related work in Section 7.2. The system model and log interface follows in Section 7.3. We then present a set of use cases of Chariots in Section 7.4. These are data management and analytics applications. The detailed design of the log is then proposed in Sections 7.5 and 7.6. Results of the evaluations are provided in Section 7.7. We conclude in Section 7.8.

Consistency	Partitioned	Replicated	systems
Strong	✓	✗	CORFU/Tango [125, 124] LogBase [93] RAMCloud [133] Blizzard [134] Ivy [135] Zebra [136] Hyder [92]
Strong	✗	✓	Megastore [37] Paxos-CP [33]
Causal	✗	✓	Message Futures [7] PRACTI [130] Bayou [137] Lazy Replication [129] Replicated Dictionary [87]
Causal	✓	✓	Chariots

Table 7.1: Comparison of different shared log services based on consistency guarantees, support of per-replica partitioning, and replication.

7.2 Related Work

In this chapter, we propose a geo-replicated shared log service for data management called Chariots. Here we briefly survey related work and how Chariots fits in the literature. We focus on systems that manage shared logs. There exist an enormous amount of work on general distributed (partitioned) storage and geo-replication. Our focus on work tackling shared logs stems from the unique challenges that shared logs pose compared to general distributed storage and geo-replication. We provide a summary of shared log services for data management application in Table 7.1. In the remainder of this section, we start with an overview of the use of logs for data management systems. Then, we provide more details about shared log systems in addition to other related work that do not necessarily provide log interfaces. We conclude with a discussion of the comparison

provided in Table 7.1.

7.2.1 Immutability and logs for data management

The use of immutability and the "append-only" style of programming was explored by various early data management systems [138, 139]. R [138] uses shadow paging. When an update is performed, a copy (shadow) of the updated page is created. The update is performed in the shadow page. POSTGRES [139] employs delta records that represent changes to the state of the system. Immutability is explored at different contexts as well. Edelweiss [123] uses Events Log Exchange to coordinate between participants in a distributed computation. The log abstraction is used widely for data management applications durability by variants of ARIES [140] recovery and Write-ahead logging (WAL) algorithm. However, the log in ARIES is used for recovery only and is not exposed to the application client. In addition, even if the log is exposed to clients, WAL techniques maintain data in two forms, one for manipulation and one for persistence. This means that writes are performed twice and data representation need to be transformed, which poses an additional overhead. The log abstraction is also used widely for Log-structuring [141].

7.2.2 Partitioned shared logs

Several systems explored extending shared logs as a distributed storage spanning multiple machines. Hyder [92] builds a multi-version log-structured database on a distributed shared log storage. A transaction executes optimistically on a snapshot of the database and broadcasts the record of changes to *all* servers and appends a record of changes to the distributed shared log. The servers then commit the transaction by looking for conflicts in the shared log in an intelligible manner. LogBase [93], which is similar to network filesystems like BlueSky [142], and RAMCloud [133] are also multi-version

log-structured databases.

Corfu [124], used by Tango [125], attempts to increase the throughput of shared log storage by employing a sequencer. The sequencer's main function is to pre-assign log position ids for clients wishing to append to the log. This increases throughput by allowing more concurrency. However, the sequencer is still a bottleneck limiting the scalability of the system.

Distributed and networked filesystems also employ logs to share their state. Blizzard [134] proposes a shared log to expose a cloud block storage. Blizzard decouples ordering and durability requirements, which improves its performance. Ivy [135] is a distributed file system. A log is dedicated to each participant and is placed in a distributed hash table. Finding data requires consulting all logs but appending is done to the participant's log only. The Zebra file system [136] employs log striping across machines to increase throughput.

7.2.3 Replicated shared logs

Causal replication. Causal consistency for availability is utilized by various systems [42, 129, 130, 131, 137, 129]. Recently, COPS [42] proposes causal+ consistency that adds convergence as a requirement in addition to causal consistency. COPS design aims to increase the throughput of the system for geo-replicated environments. At each datacenter, data is partitioned among many machines to increase throughput. Chariots targets achieving high throughput similarly by scaling out. Chariots differs in that it exposes a log rather than a key-value store, which brings new design challenges. Logs have been utilized by various replication systems for data storage and communication. PRACTI [130] is a replication manager that provides partial replication, arbitrary consistency, and topology independence. Logs are used to exchange updates and invalidation

information to ensure the storage maintains a causally consistent snapshot. Bayou [137] is similar to PRACTI. In Bayou, batches of updates are propagated between replicas. These batches have a start and end times. When a batch is received, the state rolls back to the start time, incorporate the batch, and then roll forward the existing batches that follows. Replicated Dictionary [87] replicates a log and maintains causal relations. It allows transitive log shipping and maintains information about the knowledge of other replicas. Lazy Replication [129] also maintains a log of updates ordered by their causal relations. The extent of knowledge of other replicas is also maintained.

Geo-replicated logs. Geo-replication of a shared log has been explored by few data management solutions. Google megastore [37] is a multi-datacenter transaction manager. Megastore commit transactions by contending for log positions using Paxos [26]. Paxos-CP [33] use the log in a similar way to megastore with some refinements to allow better performance. These two systems however, operate on a serial log. All clients contend to write to the head of the log, making it a single point of contention, which limits throughput. Message Futures [7] and Helios [8] are commit protocols for strongly consistent transactions on geo-replicated data. They build their transaction managers on top of a causally-ordered replicated log that is inspired from Replicated Dictionary [87].

7.2.4 Summary and comparison

The related works above that build shared logs for data management applications are summarized in Table 7.1. We display whether the system support partitioning and replication in addition to the guaranteed ordering consistency. Consistency is either strong, meaning that the order is either identical or serializable for replicated logs or totally ordered for non-replicated logs. A system is *partitioned* if the shared log spans more than one machine for each replica. Thus, if a system of five replicas consists of five

machines, they are *not* partitioned. A system is *replicated* if the shared log has more than one *independent* copy.

Other than Chariots, the table lists four systems that support partitioning. It is possible for these systems to employ replication in the storage level. However, a straight-forward augmentation of a replication solution will be inefficient. This is because a general-purpose replication method will have guarantees stronger than what is needed to replicate a log. The other solutions, that support replication, do not support partitioning. Handling a replica with a single node limits the achievable throughput. The processing power, I/O, and communication of a single machine can not handle the requirements of todays web applications. This is specially true for geo-replication that handles datacenter-scale demand.

Chariots attempts to fill this void of shared logs that have both a native support of replication and per-replica partitioning. This need for both replication and partitioning has been explored for different applications and guarantees, including causal consistency, *i.e.*, COPS [42]. However, geo-replication of a distributed shared log and immutable updating pose unique challenges that are not faced by geo-replication of key-value stores and block-based storage. The chapter studies these challenges and design Chariots as a causally-ordered shared log that supports per-replica partitioned log storage and geo-replication.

7.3 System and programming model

Chariots is a shared log system for cloud applications. The inner workings of Chariots are not exposed to the application developer. Rather, the developer interacts with Chariots via a set of APIs. In this section, we will show the interface used by developers to write applications using Chariots.

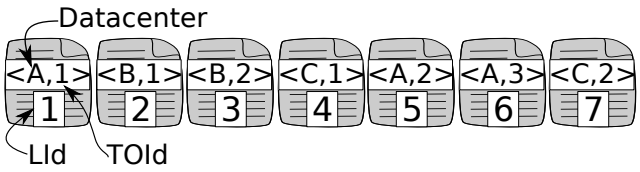


Figure 7.2: Records in a shared log showing their TOId inside the records alongside the datacenters that created them and the records LIds under the log

System model. Chariots exposes a log of records to applications. The log is maintained by a group of machines called the *log maintainers*. Collectively, these log maintainers persist a single shared log. Each log maintainer is responsible for a disjoint range of the shared log. The shared log is accessed by cloud applications, called *application clients*, through a linked library that manages the exchange of information between the application and the log maintainers. Application clients are distributed and independent from one another. And they share a single view of the shared log. The shared log is fully replicated to a number of datacenters. In our model, we adopt a view of the *datacenter as a computer* [143], an increasingly popular view of computing that reflects the demand of datacenter-scale applications.

Meta information about log maintainers, other datacenters, and the shared log are managed by *meta servers*. Meta servers are a highly-available collection of stateless servers acting as an oracle for application clients to report about the state and locations of the Log maintainers and other datacenters. This model of scale-out distributed computing and centralized stateless highly-available control servers has been shown to perform the best for large-scale systems [73].

Data model. The state of the shared log consists of the records it contains. These records are either *local copies*, meaning that they were generated by application clients residing in the same datacenter, or *external copies*, meaning that they were generated at other datacenters. Each record has an identical copy at each datacenter, one of which is considered a local copy and the other copies are considered external copies. The record

consists of the contents appended by the Application client, called the record's *body*, and other *meta-information* that are used by Application clients to facilitate future access to it. The following are the meta-information maintained for each record:

- *Log Id (LId):* This id reflects the position of the record in the datacenter where the copy resides. A record has multiple copies, one at each datacenter. Each copy has a different LId that reflects its position in the datacenter's shared log.
- *Total order Id (TOId):* This id reflects the total order of the record with respect to its host datacenter, where the Application client that created it resides. Thus, copies of the same record have an identical TOId.
- *Tags:* The Application client might choose to attach tags to the record. A tag consists of a key and a value. These tags are accessible by Chariots, unlike the record's body which is opaque to the system. Records will be indexed using these tags.

To highlight the difference between LId and TOId, observe the sample shared log in Figure 7.2. It displays seven records with their LIds in the bottom of each record at datacenter A. The TOId is shown inside the record via the representation $< X, i >$, where X is the host datacenter of the Application client that appended the record and i is the TOId. Each record has a LId that reflects its position in the shared log of datacenter A. Additionally, each record has a TOId that reflects its order compared to records coming from the same datacenter only.

Programming interface and model. Application clients can observe and change the state of the shared log through a simple interface of two basic operations: reading and appending records. These operations are performed via an API provided by a linked software library at the Application client. The library needs the information of the meta

servers only to initiate the session. Once the session is ready, the application client may use the following library calls:

1. Append(in: record, tags): Insert record to the shared log with the desired tags. The assigned TOId and LId will be sent back to the Application client. Appended records are automatically replicated to other replicas.

2. Read(in: rules, out: records): Return the records that matches the input rules. A rule might involve TOIds, LIds, and tags information.

Log records are immutable, meaning that once a record is added, it cannot be modified. If an application client desire to alter the effect of a record it can do so by appending another record that exemplifies the desired change. This principle of accumulation of changes, represented by immutable data, is identified to reduce the problems arising from distributed programming [120, 121, 122]. Taking this principled approach and combining it with the simple interface of appends and reads allows the construction of complex software while reducing the risks of distributed programming. We showcase the potential of this simple programming model by constructing data management systems in the next section.

Causality and log order. The shared log at each datacenter consists of a collection of records added by application clients at different datacenters. Ordering the records by causality relations allows sufficient consistency while preserving availability and fault-tolerance [126, 127]. Causality enforces two types of order relations [41] between read and append operations, where $o_i \rightarrow o_j$ denotes that o_i has a causal relation to o_j. A causal relation, $o_i \rightarrow o_j$, exists in the following cases:

- **Total order** for records generated from the same datacenter. If two appended records, o_i and o_j, were generated by application clients residing in the same

datacenter A, then if o_i is ordered before o_j in A, then this order must be honored at all other datacenters.

- **Happened-before relations** between read and append operations. A happened-before relation exists between an append operation, o_i, and a read operation, o_j, if o_j reads the record appended by o_i.
- **Transitivity:** causal relations are transitive. If a record o_k exists such that $o_i \rightarrow o_k$ and $o_k \rightarrow o_j$ then $o_i \rightarrow o_j$.

7.4 Case studies

The simple log interface was shown to enable building complex data management systems [33, 37, 92, 7, 123, 93]. These systems, however, operate on a serial log with pre-assigned log positions. These two characteristics, as we argued earlier, limits the log's availability and scalability. In this section, we demonstrate data management systems that are built on top of Chariots, a causally ordered log with post-assigned log positions. The first system, *Hyksos*, is a replicated key-value store that provides causal consistency with a facility to perform *get transactions*. The second system is a stream processor that operates on streams originating from multiple datacenters. We also refer to our earlier work, Message Futures [7] and Helios [8], which provide strongly consistent transactions on top of a causally ordered replicated log. Although they were introduced with a single machine per replica implementation, their design can be extended to be deployed on the scalable Chariots.

7.4.1 Hyksos: causally consistent key-value store

Hyksos is a key-value store built using Chariots to provide causal consistency [41]. Put and Get operations[1] are provided by Hyksos in addition to a facility to perform get transactions (GET_TXN) of multiple keys. Get transactions return a consistent state snapshot of the read keys.

Design and algorithms

Chariots manages the shared log and exposes a read and append interface to application clients, which are the drivers of the key-value store operations. Each datacenter runs an instance of Chariots. An instance of Chariots is comprised of a number of machines. Some of the machines are dedicated to store the shared log and others are used to deploy Chariots.

The value of keys reside in the shared log. A record holds one, or more put operation information. The order in the log reflects the causal order of put operations. Thus, the current value of a key, k, is in the record with the highest log position containing a put operation. The get and put operations are performed as follows:

- Get(x): Perform a Read operation on the log. The read returns a recent record containing a put operation to x.
- Put(x, value): Putting a value is done by performing an Append operation with the new value of x. The record must be tagged with the key and value information to enable an efficient get operation.

Get transactions. Hyksos provides a facility to perform get transactions. The get_transaction operation returns a consistent view of the key-value store. The application

[1]The terms "read" and "append" are used for operations on the log and "put" and "get" are used for operations on the key-value store.

Algorithm 14: Performing Get_transactions in Hyksos

```
1: // Request the head of the log position id
2: i = get_head_of_log()
3: // Read each item in the read set
4: for each k in read-set
5:     t = Read ({tag: k, LId<i}, most-recent)
6:     Output.add (t)
```

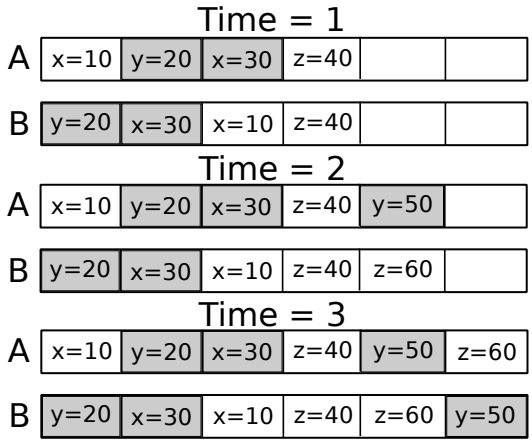


Figure 7.3: An example of Hyksos, the key-value store built using Chariots.

client performs the Get operations as shown in Algorithm 14. First, Chariots is polled to get the head of the log's position id, i, to act as the log position when the consistent view will be taken (Line 2). There must be no gaps at any records prior to the log id. Afterwards, the application client begins reading each key k (Lines 4-6). A request to read the version of k at a log position j that satisfies the following: Record j contains the most recent write to k that is at a position less than i.

Example scenario

To demonstrate how Hyksos works, consider the scenario shown in Figure 7.3. It displays the shared logs of two datacenters, A and B. The shared log contains records of put operations. The put operation is in the form "$x = v$", where x is the key and v is the value. Records that are created by Application clients at A are shaded. Other records are

created by application clients at B.

The scenario starts with four records, where each record has two copies, one at each datacenter. Two of these records are put operations to key x. The other two operations are a put to y and a put to z. The two puts to x were created at different datacenters. Note that the order of writes to x is different at A and B. This is permissible if no causal dependencies exist between them. At time 1, a Get of x at A will return 30, while 10 will be returned if the Get is performed at B.

At time 2, two Application clients, one at each datacenter, perform put operations. At A, Put(y,50) is appended to the log. At B, Put(z,60) is appended to the log. Now consider a get transaction that requests to get the value of x, y and z. First, a non-empty log position is chosen. Assume that the log position 4 is chosen. If the get transaction ran at A, it will return a snapshot of the view of the log up to log position 4. This will yield $x = 30$, $y = 20$, and $z = 40$. Note that although a more recent y value is available, it was not returned by the get transactions because it is not part of the view of records up to position 4. If the get transaction ran at B, it will return $x = 10$, $y = 20$, and $z = 40$.

Time 3 in the figure shows the result of the propagation of records between A and B. $Put(y, 50)$ has a copy now at B, and $Put(z, 60)$ has a copy at A.

7.4.2 Event processing

Another application targeted by Chariots is multi-datacenter event processing. Many applications generate a large footprint that they would like to process. The users' interactions and actions in a web application can be analyzed to generate business knowledge. These events range from click events to the duration spent in each page. Additionally, social networks exhibit more complex analytics of events related to user-generated contents (*e.g.*, micro-blogs) and user-user relationships to these events. Frequently, such analytics

are carried in multiple datacenters for fault-tolerance and locality [70, 73].

Chariots enables a simple interface for these applications to manage the replication and persistence of these analytics while preserving the required exactly-once semantics. Event processing applications consist of publishers and readers. Publishing an events is as easy as performing an append to the log. Readers then read the events from the log maintainers. An important feature of Chariots is that readers can read from different log maintainers. This will allow distributing the analysis work without the need of a centralized dispatcher that can be a single-point of contention.

7.4.3 Message Futures and Helios

Message Futures [7] and Helios [8] are commit protocols that provide strongly consistent transactions on geo-replicated data stores. They leverage a replicated log that guarantees causal order [87]. A single node at each datacenter, call it replica, is responsible for committing transactions and replication. Transactions consist of read and write operations and are committed optimistically. Application clients read from the data store and buffer writes. After all operations are ready, a *commit request* is sent to the closest replica. A record is appended to the log to declare the transaction t as ready to begin the commit protocol. Message Futures and Helios implement different conflict detection protocols to commit transactions. Message Futures [7] waits for other datacenters to send their histories up to the point of t's position in the log. Conflicts are detected between t and received transactions, and t commits if no conflicts are detected. Helios [8] builds on a lower-bound proof that determines the lowest possible commit latency that a strongly consistent transaction can achieve. Helios commits a transaction t by detecting conflicts with transactions in a *conflict zone* in the shared log. The conflict zone is calculated by Helios using the lower-bound numbers. If no conflicts were detected, t commits. A full

description of Message Futures and Helios are available in previous publications [7, 8].

Message Futures and Helios demonstrate how a causally ordered log can be utilized to provide strongly consistent transactions on replicated data. However, the used replicated log solution [87] is rudimentary and is not suitable for today's applications. It only utilizes a single node per datacenter. This limits the throughput that can be achieved to that of a single node. Chariots can be leveraged to scale Message Futures and Helios to larger throughputs. Rather than a replica with a single node at each datacenter, Chariots would be used to distribute storage and computation.

7.4.4 Summary and remarks

The simple interface of Chariots enabled the design of web and analytics applications. The developer can focus on the logic of the data manager or stream processor without having to worry about the details of replication, fault-tolerance, and availability. In addition, the design of Chariots allows scalable designs of these solutions by having multiple sinks for reading and appending.

7.5 FLStore: distributed shared log

In this section, we describe the distributed implementation of the shared log, called the Fractal Log Store (FLStore). FLStore is responsible of maintaining the log *within* the datacenter. We begin by describing the design of the distributed log storage. Then, we introduce the scalable indexing component used for accessing the shared log.

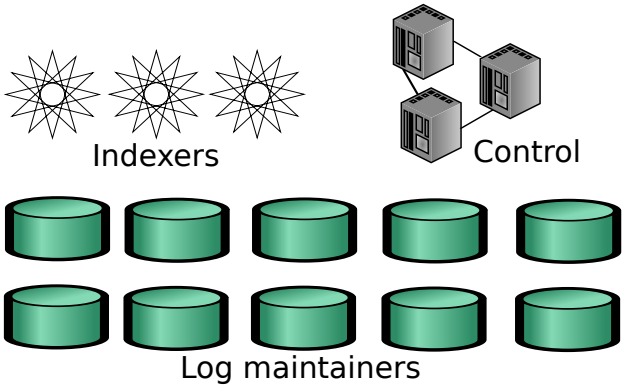


Figure 7.4: The architecture of FLStore

7.5.1 Architecture

In designing FLStore, we follow the principle of distributing computation and highly-available stateless control. This approach has been identified as the most suitable to scale out in cloud environments [73]. The architecture of FLStore consists of three types of machines, shown in Figure 7.4. *Log maintainers* are responsible for persisting the log's records and serving read requests. *Indexers* are responsible of access to log maintainers. Finally, control and meta-data management is the responsibility of a highly-available cluster called the *Controller*.

Application clients start their sessions by polling the Controller for information about the indexers and log maintainers. This information includes the addresses of the machines and the log ranges falling under their responsibility in addition to approximate information about the number of records in the shared log. Appends and reads are served by Log maintainers. The Application client communicates with the Controller only at the beginning of the session or if communication problems occur. And Application clients will communicate with Indexers only if read operation did not specify LIds in the rules.

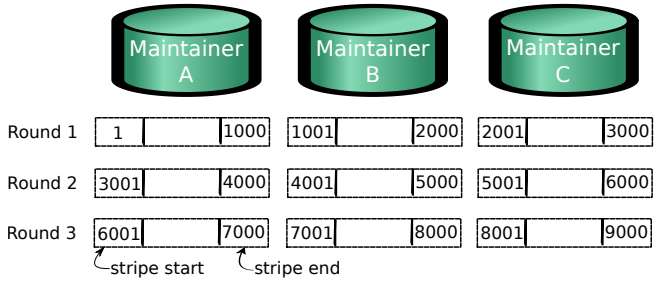


Figure 7.5: An example of three deterministic log maintainers with a batch size of 1000 record. Three rounds of records are shown.

7.5.2 Log maintainers

Scalability by post-assignment. The Log maintainers are accessed via a simple interface for adding to and reading from the shared log. They are designed to be *fully distributed* to overcome the I/O bandwidth constraints that are exhibited by current shared log protocols. A recent protocol is CORFU [124] that is limited by the I/O bandwidth of a sequencer. The sequencer is needed for CORFU to pre-assign log positions to application clients wishing to append records to the log. In FLStore, we abandon this requirement of pre-assigning log positions and settle for a *post-assignment* approach. The thesis of a post-assignment approach is to let the application client construct the record and send it to a randomly (or intelligibly) selected Log maintainer. The Log maintainer will assign the record the next available log position from log positions under its control.

Design. The shared log is distributed among the participating Log maintainers. This means that each machine holds a partial log and is responsible for its persistence and for answering requests to read its records. This distribution poses two challenges. The first is the way to append to the log while guaranteeing uniqueness and the non-existence of gaps in the log. This includes the access to these records and the way to index the records. The other challenge is maintaining explicit order guarantees requested by the application client. We employ a *deterministic* approach to make each machine responsible for specific ranges of the log. These ranges round-robin across machines where each round consists

of a number of records. we will call this number the batch size. Figure 7.5 depicts an example of three log maintainers, A, B, and C. The figure shows the partial logs of the first three rounds if the batch size was set to a 1000 records.

If an application wants to read a record it directs the request to the Log maintainer responsible for it. The Log maintainer can only answer requests of records if their LIds are provided. Otherwise, the reader must collect the LIds first from the Indexers as we show in the next section. Appending to the log is done by simply sending a record or group of records to one of the Log maintainers. The Log maintainer appends the record to the next available log position. It is possible that a log maintainer will receive more record appends than others. This creates a load-balancing problem that can be solved by giving the application feedback about the rate of incoming requests at the maintainers. This feedback can be collected by the Controller and be delivered to the application clients as a part of the session initiation process. Nonetheless, this is an orthogonal problem that can be solved by existing solutions in the literature of load balancing.

7.5.3 Distributed indexing

Records in Log maintainers are arranged according to their LIds. However, Application clients often desire to access records according to other information. When an Application client appends a record it also *tags* it with access information. These tags depend on the application. For example, a key-value store might wish to tag a record that has Put information with the key that is written. For this reason, we utilize distributed Indexers that provide access to the Log maintainers by tag information. Distributed indexing for distributed shared logs is tackled by several systems [92, 144, 93]

Tag and lookup model. The tag is a string that describes a feature of the record. It is possible that the tag also has a value. Each record might have more than one tag.

The application client can lookup a tag by its name and specify the amount of records to be returned. For example, the Application client might lookup records that has a certain tag and request returning the most recent 100 record LIds to be returned with that tag. If the tag has a value attached to it, then the Application client might lookup records with that tag and rules on the value, *e.g.*, look up records with a certain tag with values greater than i and return the most recent x records.

7.5.4 Challenges

Log gaps. A Log maintainer receiving more records advances in the log ahead of others. For example, Log maintainer A can have 1000 records ready while Log maintainer B has 900 records. This causes temporary gaps in the logs that can be observed by Application clients reading the log. The requirement that needs to be enforced is that *Application clients must not be allowed to read a record at log position i if there exist at least one gap at log position j less than i.*

To overcome the problem of these temporary gaps, minimal gossip is propagated between maintainers. The goal of this gossip is to identify the record LId that will guarantee that any record with a smaller LId can be read from the Log maintainers. We call this LId the *Head of the Log (HL)*. Each Log maintainer has a vector with a size equal to the number of maintainers. Each element in the vector corresponds to the maximum LId at that maintainer. Initially the vector is initialized to all zeros. Each maintainer updates its value in the local vector. Occasionally, a maintainer propagates its maximum LId to other maintainers. When the gossip message is received by a maintainer it updates the corresponding entry in the vector. A maintainer can decide that the HL value is equal to the vector entry with the smallest value. When an application wants to read or know the HL, it asks one of the maintainers for this value. This technique does not pose a

significant bottleneck for throughput. This is because it is a fixed-sized gossip that is not dependent on the actual throughput of the shared log. It might, however, cause the latency to be higher as the the throughput increases. This is because of the time required to receive gossip messages and determine whether a LId has no prior gaps.

Explicit order requests. Appends translate to a total order at the datacenter after they are added by the Log maintainers. Concurrent appends therefore do not have precedence relative to each other. It is, however, possible to enforce order for concurrent appends if they were requested by the Application client. One way is to send the appends to the same maintainer in the order wanted. Maintainers ensure that a latter append will have a LId higher than ones received earlier. Otherwise, it is possible to enforce order for concurrent appends across maintainers. The Application client waits for the earlier append to be assigned a LId and then attach this LId as a minimum bound. The maintainer that receives the record with the minimum bound ensures that the record is buffered until it can be added to a partial log with LIds larger than the minimum bound. This solution however must be pursued with care to avoid a large backlog of partial logs.

7.6 Chariots: geo-replicated log

In this section we show the design of Chariots that supports multi-datacenter replication of the shared log. The design is a multi-stage pipeline that includes FLStore as one of the stages. We begin the discussion by outlining an abstract design of log replication. This abstract design specifies the requirements, guarantees, and interface desired to be provided by Chariots. The abstract solution will act as a guideline in building Chariots, that will be proposed after the abstract design. Chariots is a distributed scale-out platform to manage log replication with the same guarantees and requirements of the abstract solution.

7.6.1 Abstract solution

Before getting to the distributed solution, it is necessary to start with an efficient **abstract solution**. This abstract solution will be provided here in the form of algorithms running on a totally ordered thread of control at the datacenter. This is similar to saying that the datacenter is the machine and it is manipulating the log according to incoming events. Using this abstract solution, we will design the distributed implementation next (Section 7.6.2) that will result in a behavior identical to the abstract solution with a higher performance.

The data structures used are a log and a $n \times n$ table, where n is the number of datacenters, called the Awareness Table (ATable) inspired by Replicated Dictionary [87]. The table represents the datacenter's (DC's) extent of knowledge about other DCs. Each row or column represents one DC. Consider DC A and its ATable T_A. The entry $T_A[B, C]$ contains a TOId, τ, that represents B's knowledge about C's records according to A. This means that A is certain that B knows about all records generated at host DC C up to record τ. When a record is added to the log, it is tagged by the following information: (1) *TOId*, (2) *Host datacenter Id*, and (3) causality information.

The body of the record, which is supplied by the application is opaque to Chariots . To do the actual replication, the local log and ATable are continuously being propagated to other DCs. When the log and ATable are received by another DC, the new records are incorporated at the receiving log and the ATable is updated accordingly.

The algorithms to handle operations and propagation are presented with the assumption that only one node manipulates Chariots, containing the log of records and ATable. In Section 7.6.2 we will build the distributed system that will be manipulating Chariots while achieving the correct behavior of the abstract solution's algorithms presented here. The following are the events that need to be handled by Chariots:

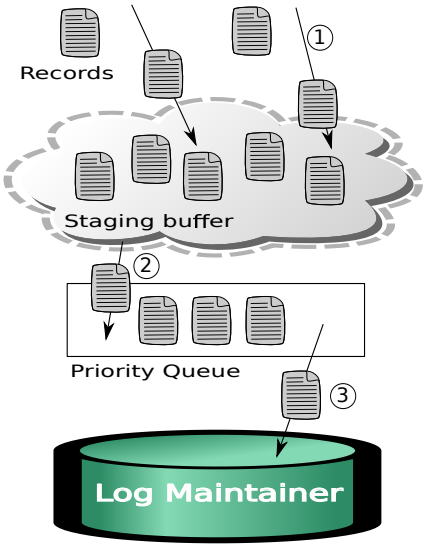


Figure 7.6: The components involved in adding records in the abstract solution.

1. **Initialization**: The log is initialized and the ATable entries are set to zero. Note that the first record of each node has a TOId of 1.

2. **Append**: Construct the record by adding the following information: host identifier, TOId, LId, causality, and tags. Update the entry $T_I[I, I]$, where I is the host datacenter's id, to be equal to the record's TOId. Finally, add the record to the log.

3. **Read**: Get the record with the specified LId.

4. **Propagate**: A snapshot of Chariots is sent to another DC j. The snapshot includes a subset of the records in the log that are not already known by j. Whether a record, r, is known to j can be verified using $T_i[j, i]$ and comparing it to $TOId(r)$.

5. **Reception**: When a log is received, incorporate all the records that were not seen before to the local log if its causal dependencies are satisfied. Otherwise, add the record with unsatisfied dependencies to a priority queue ordered according to causal relations. This is depicted in Figure 7.6. The incoming records are all put in a staging buffer (step 1) and are taken and added to the log or priority queue according

to their causal dependencies (step 2). Chariots checks the priority queue frequently to transfer any records that have their dependencies satisfied to the log (step 3). Also, the ATable is updated to reflect the newly incorporated records.

Garbage collection. The user has the choice to either garbage collect log records or maintain them indefinitely. Depending on the applications, keeping the log can have great value. If the user choses not to garbage collect the records then they may employ a cold storage solution to archive older records. On the other hand, the user can choose to enable garbage collection of records. It is typical to have a temporal or spatial rule for garbage collecting the log. However, in addition to any rule set by the system designer, garbage collection is performed for records only after they are known by all other replicas. This is equivalent to saying that a record, r, can be garbage collected at i if and only if $\forall_{j \in nodes}(T_i[j, host(r)] \geq ts(r))$, where $host(r)$ is the host node of r.

7.6.2 Chariots distributed design

In the previous section we showed an efficient abstract design for a shared log that supports multi-datacenter replication. Chariots is a distributed system that mimics that abstract design. Each datacenter runs an instance of Chariots. The shared logs at different datacenters are replicas. All records exist in all datacenters. The system consists of a multi-stage pipeline. Each stage is responsible of performing certain tasks to incoming records and pushing them along the pipeline where they eventually persist in the shared log. Each stage in Chariots is designed to be elastic. An important design principle is that Chariots is designed to identify bottlenecks in the pipeline and allow overcoming them by adding more resources to the stages that are overwhelmed. For this to be successful, elasticity of each stage is key. Minimum to no dependencies exist between the machines belonging to one stage.

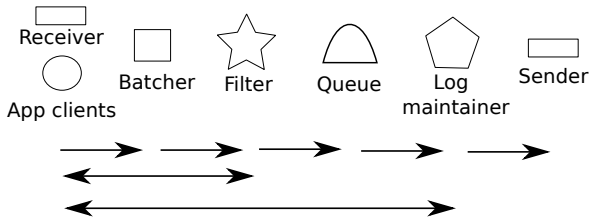


Figure 7.7: The components of the multi-data center shared log. Arrows denote communication pathways in the pipeline.

Pipeline design. Chariots pipeline consists of six stages depicted in Figure 7.7. The first stage contains nodes that are generating records. These are Application clients and machines receiving the records sent from other datacenters. These records are sent to the next stage in the pipeline, *Batchers*, to batch records to be sent collectively to the next stage. *Filters* receive the batches and ensure the uniqueness of records. Records are then forwarded to Queues where they are assigned LId. After assigning a LId to a record it is forwarded to FLStore that constitutes the Log maintainers stage. The local records in the log are read from FLStore and sent to other datacenters via the *Senders*.

The arrows in Figure 7.7 represent the flow of records. Generally, records are passed from one stage to the next. However, there is an exception. Application clients can request to read records from the Log maintainers. Chariots support elastic expansion of each stage to accommodate increasing demand. Thus, each stage can consist of more than one machine, *e.g.*, five machines acting as Queues and four acting as Batchers. The following is a description of each stage:

Application clients. The Application client hosts the application modules. These modules uses the interface to the log that was presented in Section 7.3. Some of the commands are served by only reading the log. These include Read and control commands. These requests are sent directly to the Log maintainers. The Append operation creates a record that encapsulates the user's data and send it to any Batcher machine.

Batchers. The Batchers buffer records that are received locally or from external

sources. Batchers are completely independent from each other, meaning that no communication is needed from one Batcher to another and that scaling to more batchers will have no overhead. Each Batcher has a number of buffers equal to the number of Filters. Each record is mapped to a specific Filter to be sent to it eventually. Once a buffer size exceeds a threshold, the records are sent to the designated Filter. The way records are mapped to Filters is shown next.

Filters. The Filters ensures uniqueness of records. To perform this task, each Filter becomes a champion for a subset of the records. One natural way to do so is to make each Filter a champion for records with the same host Id, i.e. the records that were created at the same datacenter. If the number of Filters needed is less that the number of datacenters, then a single Filter can be responsible for more than one datacenter. Otherwise, if the number of needed Filters is in fact larger than the number of datacenters, then more than one Filter need to be responsible for a single datacenter's records. For example, consider that two Filters, x and y, responsible for records coming from datacenter A. Each one can be responsible for a subset of the records coming from A. This can be achieved by leveraging the unique, monotonically increasing, TOIds. Thus, x can be responsible for ensuring uniqueness of A's records with odd TOIds and y can ensure the uniqueness of records with even TOIds. Of course, any suitable mapping can be used for this purpose. To ensure uniqueness, the processing agent maintains a counter of the next expected TOId. When the next expected record arrives it is added to the batch to be sent to the one of the Queues. Note also that this stage does not require any communication between filters, thus allowing seamless scalability.

Queues. Queues are responsible for assigning LIds to the records. This assignment must preserve the ordering guarantees of records. To append records to the shared log they need to have all their causal dependencies satisfied in addition to the total order of records coming from the same datacenter. Once a group of records have their causal

dependencies satisfied, they are assigned LIds and sent to the appropriate log maintainer for persistence. For multi-datacenter records with causal dependencies, it is not possible to append to the FLStore directly and make it assign LIds in the same manner as the single-datacenter deployment shown in section 7.5.2. This is because it is not guaranteed that any record can be added to the log at any point in time, rather, its dependencies must be satisfied first. The queues ensure that these dependencies are preserved and assign LIds for the records before they are sent to the log maintainers. The queues are aware of the deterministic assignment of LIds in the log maintainers and forward the records to the appropriate maintainer accordingly.

Queues ensure causality of LId assignments by the use of a token. The token consists of the current maximum TOId of each datacenter in the local log, the LId of the most recent record, and the deferred records with unsatisfied dependencies. The token is initially placed at one of the Queues. The Queue holding the token append all the records that can be added to the log. The Queue can verify whether a record can be added to the shared log by examining the maximum TOIds in the token. The records that can be added are assigned LIds and sent to the Maintainers designated for them. The token is updated to reflect the records appended in the log. Then, the token is sent to the next maintainer in a round-robin fashion. The token might include all, some, or none of the records that were not successfully added to the log. Including more deferred records with the token consumes more network I/O. On the other hand, not forwarding the deferred records with the token might increase the latency of appends. It is a design decision that depends on the nature of Chariots deployment.

Log maintainers. These Log maintainers are identical to the distributed shared log maintainers of FLStore presented in Section 7.5.2. Maintainers ensure the persistence of records in the shared log. The record is available to be read by senders and application clients when they are persisted in the maintainers.

Log propagation. Senders propagate the local records of the log to other datacenters. Each sender is limited by the I/O bandwidth of its network interface. To enable higher throughputs, more Senders are needed at each datacenter. Likewise, more Receivers are needed to receive the amount of records sent. Each Sender machine is responsible to send parts of the log from some of the maintainers to a number of Receivers at other datacenters.

7.6.3 Live elasticity

The demand on web and cloud applications vary from time to time. The ability of the infrastructure to scale to the demand seamlessly is a key feature for its success. Here, we show how adding compute resources to Chariots in the fly is possible without disruptions to the Application clients. The elasticity model of Chariots is to treat each stage as an independent unit. This means that it is possible to add resources to a single stage to increase its capacity without affecting the operation of other stages.

Completely independent stages. Increasing the capacity of completely independent stages merely involves adding the new resources and sending the information of the new machine to the higher layer. The completely independent stages are the receivers, batchers, and senders. For adding a receiver, the administrator needs to inform senders of other datacenters so that it can be utilized. Similarly, a new batcher need to inform local receivers of its existence. A new sender is different in that it is the one reading from log maintainers, thus, the log maintainers need not be explicitly told about the introduction of a new sender.

Filters. In Chariots, each filter is championing a specific subset of the log records. Increasing the number of filters results in the need of reassigning championing roles. For example, a filter that was originally championing records from another datacenter

could turn out to be responsible for only a subset of these records while handing off the responsibility of the rest of them to the new filter. This reassignment need to be orchestrated with batchers. There need to be a way for batchers to figure out when the hand-over took place so that they can direct their records accordingly. A *future reassignment* technique will be followed for filters as well as log maintainers as we show next. A future reassignment for filters begin by marking future TOIds that are championed by the original filter. These future TOIds mark transition of championing a subset of the records to the new filter. Consider a filter that champions records from datacenter A in a reassignment scenario of adding a new filter that will champion the subset of these records with even TOIds. Marking a future TOId, t, will result in records with even TOIds greater than t to be championed by the new filter. This future reassignment should allow enough time to propagate this information to batchers.

Queues. Adding a new queue involves two tasks: making the new queue part of the token exchange loop and propagating the information of its addition to filters. The first task is performed by informing one of the queues that it should forward the token to the new queue rather than the original neighbor. The latter task (informing filters) can be performed without coordination because a queue can receive any record.

Log maintainers. Expanding log maintainers is similar to expanding filters in that each maintainer champions a specific set of records. In this case, each log maintainer champions a subset of records with specific LIds. The future reassignment technique is used in a similar way to expanding filters. In this case, not only do the queues need to know about the reassignment, but the readers need to know about it too. Another issue is that log maintainers persist old records. Rather than migrating the old records to the new champion, it is possible to maintain an epoch journal that denotes the changes in log maintainer assignments. These can be used by readers to figure out which log maintainer to ask for an old record.

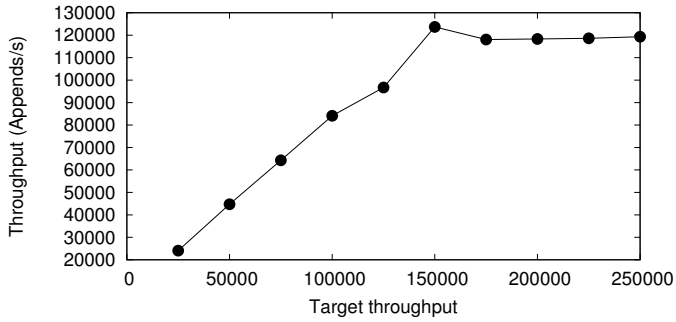


Figure 7.8: The throughput or one maintainer while increasing the load in a public cloud

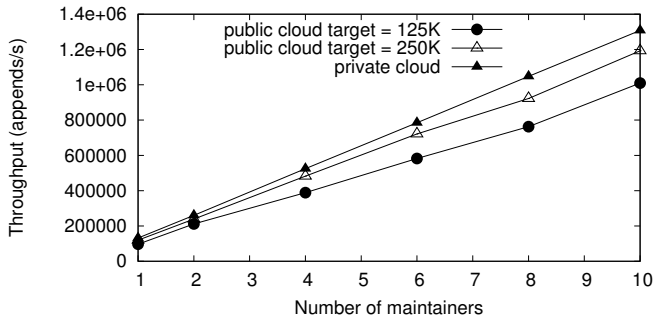


Figure 7.9: The append throughput of the shared log in a single-datacenter deployment while increasing the number of Log Maintainers.

7.7 Evaluation

In this section, we present some experiments to evaluate our implementation of FLStore and Chariots. The experiments were conducted on a cluster with nodes with the following specifications. Intel Xeon E5620 CPUs that are running 64-bit CentOS Linux with OpenJDK 1.7 were used. The nodes in a single rack are connected by a 10GB switch with an average RTT of 0.15 ms. We also perform the baseline experiments on Amazon AWS. There, we used compute optimized machines (c3.large) in the datacenter in Virginia. Each machine has 2 virtual CPUs and a 3.75 GiB memory. We refer to the earlier setup as the *private cloud* and the latter setup as the *public cloud*. Unless it is mentioned otherwise, the size of each record is 512 Bytes.

7.7.1 FLStore scalability

The first set of experiments that we will present is of the FLStore implementation which operates within the datacenter. Each one of the experiments consists of two types of machines, namely Log maintainers and clients. The clients generate records and send them to the Log Maintainers to be appended. We are interested in measuring the scaling behavior of FLStore while increasing the number of maintainers. We begin by getting a sense of the capacity of the machines. Figure 7.9 shows the throughput of one maintainer in the public cloud while increasing the load on it. Records are generated with a specific rate at each experiment point from other machines. The rate is called the *target throughput*. Note how as the target throughput increases, the achieved throughput increases up to a point and then plateaus. The maximum throughput is achieved when the target throughput is 150K and then drops to be around 120K appends per second. These numbers will help us decide what target throughputs to choose for our next experiments.

To verify the scalability of FLStore, Figure 7.9 shows the cumulative throughput of different scenarios each with a different number of maintainers. For each experiment an identical number of client machines were used to generate records to be appended. Ideally, we would like the throughput to increase linearly with the addition of new maintainers. Three plots are shown, two from the public cloud and one from the private cloud. The ones from the public cloud differ in the target throughput to each maintainer. One targets 125K appends per second for each maintainer while the other targets 250K appends per second. Note how one is below the plateau point and one is above. The figure shows that FLStore scales with the addition of resources. A single maintainer has a throughput of 131K for the private cloud, 96.7K for the public cloud with a target of 125K, and 119K for the public cloud with the target of 250K. As we are increasing the number of Log Maintainers a near-linear scaling is observed. For ten Log Maintainers, the achieved

Machine	Throughput (Kappends/s)
Client	129
Batcher	129
Filter	129
Maintainer	124
Store	132

Table 7.2: The throughput of machines in a basic deployment of Chariots with one machine per stage.

Machine	Throughput (Kappends/s)
Client 1	120
Client 2	122
Batcher	126
Filter	125
Maintainer	123
Store	132

Table 7.3: The throughput of machines in a deployment of Chariots with two clients and one machine per stage for the remaining stages.

append throughput was 1308034 record appends per second for the private cloud. This append throughput is 99.3% when compared to a perfect scaling case. The public cloud case with a target of 125K achieves a throughput that is slightly larger than the perfect scaling case. This is due to the variation in machines' performances. The other public cloud case achieve a scaling of 99.9%. This near-perfect scaling of FLStore is encouraging and demonstrates the effect of removing any dependencies between maintainers.

7.7.2 Chariots scalability

The full deployment of Chariots that is necessary to operate in a multi-datacenter environment consists of five stages. These stages are described in Section 7.6.2. Here, we will start from a basic deployment of one machine per stage in the private cloud. We observe the throughput of each stage to try to identify the bottleneck. Afterward,

Machine	Throughput (Kappends/s)
Client 1	126
Client 2	129
Batcher 1	149
Batcher 2	129
Filter	120
Maintainer	118
Store	121

Table 7.4: The throughput of machines in a deployment of Chariots with two client machines, two Batchers, and a single machine for the remaining stages.

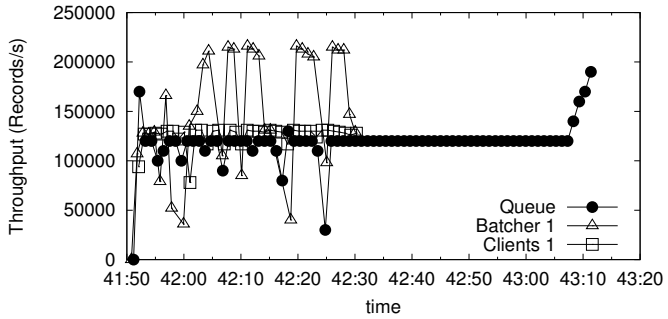


Figure 7.10: The throughput of machines in a deployment of Chariots with two client machines, two Batchers, and a single machine for the remaining stages

we observe how this bottleneck can be overcome by increasing resources. The simple deployment of one machine per stage of Chariots pipeline achieves the throughputs shown in Table 7.2. The table lists the throughput in Kilo records per second for each machine in the pipeline. Note how all machines achieve a similar throughput of records per second. It is possible for the store to achieve a throughput higher than the client because of the effect of buffering. Close throughput numbers for all machines indicates that the bottleneck is possibly due to the clients. The clients might be generating less records per second than what can be handled by the pipeline.

To test this hypothesis we increase the number of machines generating records to two client machines. The results are shown in Table 7.3. If the clients were indeed not generating enough records to saturate the pipeline, then we should observe an increase

Machine	Throughput (Kappends/s)
Client 1	130
Client 2	130
Batcher 1	127
Batcher 2	127
Filter 1	127
Filter 2	126
Maintainer 1	125
Maintainer 2	126
Store 1	137
Store 2	137

Table 7.5: The throughput of machines in a deployment of Chariots with two machines per stage.

in the throughput of the Batcher. However, this was not the case. The increased load actually resulted in a lower throughput for the batcher. This means that the batcher is possibly the bottleneck. So, we increase the number of batchers to observe the throughput of latter stages in the pipeline. Table 7.4 shows the throughput of machines with two client machines, two batchers, and a single machine for each of the remaining stages. Both batchers achieve a throughput that is higher than the one achieved by a single batcher in the previous experiments. This means that the throughput of the Batcher stage more than doubled. However, now the bottleneck is pushed to the filter stage that is not able to handle more than 130000 records per second. Because the throughput of latter stages is almost half the throughput of the Batcher, they take twice the time to finish the amount of records generated by the clients (10000000 records). The throughput timeseries for one client, one batcher, and the queue are shown in Figure 7.10. We did not show all the machines' throughputs to avoid cluttering the figure. The Batchers are done with the records at time 42:30, whereas, the latter stages lasted till time 43:10. Note that by the end of the experiment, the throughput of the queue increases abruptly. The reason for this increase is that the although the Batchers had already processed the records they are

still transmitting them to the Filter until time 43:08, right before the abrupt increase. The network interface's I/O of the Filter was limiting its throughput. After it is no longer receiving from the two Batchers it can send with a higher capacity to the latter stages, thus causing an increase in the observed throughput. This is also the reason why in the beginning of the experiment, a higher throughput is observed for some of the stages (*e.g.*, the high throughput in the beginning for the queue). The reason is that they still had capacity in their network's interface I/O before it was also used to propagate records to latter stages. Another interesting observation is the performance variation of the batcher. This turned out to be a characteristic of machines at a stage generating more throughput than what can be handled by the next stage.

Increasing the number of machines further should yield a better throughput. We experiment with the previous setting, but this time with two machines for all stages. The throughput values records are presented in Table 7.5. Note how all stages are scaling. The throughput of each stage has doubled. Each machine achieves a close throughput to the basic case of a pipeline with one machine per stage.

Our main objective is to allow scaling of shared log systems to support today's applications. We showed in this evaluation how a FLStore deployment is able to scale while increasing the number of maintainers within the datacenter (Figure 7.9). Also, we evaluated the full Chariots pipeline that is designed to be used for multi-datacenter environments. The bottleneck of a basic deployment was identified and Chariots overcomes it by adding more resources to the pipeline.

7.8 Conclusion

In this chapter, we presented a shared log system called Chariots. The main contribution of Chariots is the design of a distributed shared log system that is able to scale

beyond the limit of a single node. This is enabled by a deterministic post-assignment approach of assigning ranges of records to Log maintainers. Chariots also increases the level of availability and fault-tolerance by supporting geo-replication. A novel design to support a shared log across datacenters is presented. Causal order is maintained across records from different datacenters. To allow scaling such a shared log, a multi-stage pipeline is proposed. Each stage of the pipeline is designed to be scalable by minimizing the dependencies between different machines. An experimental evaluation demonstrated that Chariots is able to scale with increasing demand by adding more resources.

Part IV

Conclusion

Chapter 8

Summary and Concluding Remarks

Global-Scale Data Management (GSDM) has been increasingly adopted over the past several years. The opportunities enabled by GSDM—both for performance and fault-tolerance—make us confident that this is a continuing trend. As more data management systems are built with global-scalability in mind, the landscape of computing (especially cloud computing) will grow and enable applications and use cases that were not feasible before.

We believe that to make GSDM a reality, it is essential to provide systems that enable leveraging the opportunities of GSDM with accessible and easy-to-use interfaces and abstraction for programmers and developers. The work in this dissertation stems from this belief. Our goal is to enable web and cloud programmers to build their applications to benefit from the opportunities of GSDM. To achieve this goal, we considered one of the most important aspects of web and cloud applications—maintenance and access to application's state and data. Specifically, we considered transactional workloads that are essential for web and cloud applications and analytics that are important to implement Big Data applications. We focused on providing abstractions at the database layer that are easy-to-use by making them strongly consistent, thus ridding the programmer from

thinking about concurrency, replication, and other complexities related to the GSDM environment.

In this dissertation, we aspire to provide practical designs and insights that are grounded on a careful study and understanding of the fundamental characteristics of GSDM. We have found that coordination is a major obstacle in the way to implement strongly consistent abstractions efficiently. For some problems, GSDM imposes fundamental limits on the performance of strongly consistent abstractions. Understanding the nature of these limits help in designing better protocols. Each proposed protocol in this dissertation begins by thinking about the problem with wide-area coordination as the main aspect to optimize. This treatment of wide-area coordination as the main bottleneck has turned to be rewarding in terms of novel designs that achieve a much higher performance than traditional counterparts in GSDM environments. We believe that the principles we propose in these protocols will impact the design of a wide-range of problems that share the aspect of wide-area coordination.

We tackle strongly consistent transaction processing in GSDM. By focusing on the wide-area coordination aspect of the problem we have proposed a theoretical formulation of the performance limit imposed by wide-area latency in addition to a transaction management paradigm called proactive coordination. The theoretical formulation enables finding a lower-bound transaction latency in global-scale environments. This lower-bound result provides a guide to system designers and researchers in addition to a tool to reason about latency limits in environments with high communication latency. Proactive coordination is a paradigm for transaction commit protocols where coordination for future transactions start before they are issued. We have shown how this general concept can be used to implement two GSDM transaction commit protocols: (1) Message Futures, that breaks the RTT barrier for transaction latency, and (2) Helios, that is able to achieve any transaction latency that does not violate the lower-bound that we propose. Message

Futures and Helios combine traditional concurrency control approaches such as the use of timestamp ordering, log propagation, loose-time synchronization, and certification with the concepts that we propose in the dissertation such as proactive coordination and lower-bound latency. We envision that these designs will inspire future protocols that utilize proactive coordination and lower-bound latency in new, novel ways in combination with other concurrency control paradigms.

In the area of global-scale Big Data analytics and machine learning, we propose two pieces of work. Ogre is a transaction commit protocol that targets the efficient processing of analytics queries in global-scale deployments. Ogre implements a light-weight dependency tracking technique to achieve two goals: (1) Scalability by making dependency tracking at different partitions proceed independently from each other, (2) Low latency without inter-datacenter communication by utilizing dependency information to construct consistent snapshots of the database at any datacenters. We envision that the methods proposed in Ogre would be implemented over various transactional databases, as it isolates the different components of the system, which are the transaction store, analytical store, and dependency tracking components. COP is our proposal to execute machine learning tasks on data that is generated from globally-distributed sources. The main idea of COP is to leverage the opportunity when data is being collected at sources to plan execution. The plan is then used at the central machine learning workers to execute more efficiently. COP explores planning conflicts between different data samples so that they are minimized during execution. The methods proposed by COP may be used to plan other properties of execution to achieve various other advantages.

In the course of developing global-scale systems, we encountered a common challenge in managing communication between globally-distributed nodes. This motivated Chariots, our work to provide a communication platform for GSDM systems. Chariots maintains a shared log abstraction between nodes in various datacenters. Chariots targets both

scalability within the datacenter and across datacenters. Within the datacenter, Chariots includes a distributed shared log system called FLStore. FLStore removes coordination from the path of appending operations and compensate by additional asynchronous background coordination. Chariots then provides a framework to replicate distributed, shared logs across datacenters. Chariots guarantees causal order of events in the shared log, which turns out to be sufficient to implement a wide-range of systems on top of it, including Message Futures and Helios that we have proposed in the dissertation.

Chapter 9

Future Directions

Now, we overview some of the future directions that we think will be areas of active research pursuit in the coming years. These future directions are both for research to advance GSDM and research that is directly enabled, or influenced by GSDM.

In the dissertation, we have found how a lower-bound study of coordination latency for transactions is rewarding. It provided a deeper understanding of the limits of strongly consistent abstractions in GSDM in addition to inspiring new system designs that perform significantly better than previous systems. This motivates the pursuit of such theoretical foundations for GSDM. We envision that our theoretical formulation can be extended to formulate limits on different performance metrics that are related to latency (*e.g.*, throughput). Also, we envision that early theoretical work of distributed systems' limits and impossibilities will be extended to cover the new environment of GSDM. The outcome of these exercises will yield more rewards in terms of understanding GSDM and building better practical systems.

Emerging edge datacenter technologies, such as micro datacenters and cloudlets, have the potential to bring data even closer to end users. This has motivated many mobility and Internet of Things (IoT) systems to adopt the *edge computing model*, also called

by other names such as fog computing [145]. However, *for data management tasks that require updating data, many edge computing systems still rely on storage solutions that do not utilize edge datacenters.* We envision extending GSDM system beyond traditional datacenters to cover edge datacenters. Such a direction will complement and enhance existing edge computing frameworks by providing a unified storage abstraction at the edge that is globally synchronized and supports transaction processing. Edge data management enjoys the benefits—and faces the challenges—of edge computing [146]. For example, in this new model, the data management system can leverage edge datacenters for tasks such as maintaining copies for fault-tolerance and to offload smaller instances to serve requests at edge location close to users. The incorporation of edge datacenters changes the model of geo-replication. The number of replicas is no longer confined to a small number of datacenters, rather to a potentially large number of edge datacenters. Also, the Round Trip Time (RTT) between a datacenter and nearby edge datacenters is an order of magnitude lower than typical inter-datacenter RTT. Significant research and practical effort is needed to accommodate GSDM systems to this new environment. However, utilizing edge datacenters will improve the performance of GSDM systems and will enable emerging edge and mobile applications.

Even with GSDM and efficient coordination, achieving a low end-to-end latency is a challenge. This is due to the needed latency to coordinate access between different replicas to maintain consistency, and the latency to replicate data across datacenters for fault-tolerance. Reducing and controlling the latency of coordination by relaxing consistency has been the topic of many research efforts including ours. *The remaining frontier is investigating relaxing fault-tolerance to reduce and control the need for synchronous WAN communication.* We argue that analogous to how some applications may have relaxed consistency requirements, some applications may have relaxed fault-tolerance requirements. We envision an exploration of the trade-off between fault-tolerance and latency in the

context of edge data management, while preserving strongly consistent abstractions. Such exploration may lead to methods to control the fault-tolerance level in a way that result in achieving higher performance for weaker fault-tolerance levels. Also, there may be methods to relax the requirements of fault-tolerance and find a spectrum of durability guarantees with various performance characteristics.

Leveraging advances in WAN research from the networking community is an important step towards building efficient GSDM systems. Networking techniques, like Software-defined Networking (SDN), are now being applied to the context of WANs (*e.g.*, BwE [147] and B4 [148]). A promising opportunity is to develop GSDM systems that integrate these advances in networking.

Also, adopting privacy-preserving techniques for GSDM is an important endeavour to address the geo-political regulatory concerns on data usage. However, current privacy-preserving protocols are communication-intensive and rely on redundancy that cause over-utilization of network bandwidth. As such, this makes these protocols especially inefficient for GSDM. We expect a proliferation of studies on the trade-off between privacy and performance at the global scale.

Bibliography

[1] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems*. Morgan & Claypool Publishers, 1st ed., 2011.

[2] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, *Why does the cloud stop computing?: Lessons from hundreds of service outages*, in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, (New York, NY, USA), pp. 1–16, ACM, 2016.

[3] "Global internet geography." `https://www.telegeography.com/research-services/global-internet-geography/`. 2015.

[4] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, *Global analytics in the face of bandwidth and regulatory constraints*, in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, (Berkeley, CA, USA), pp. 323–336, USENIX Association, 2015.

[5] "European commission press release. commission to pursue role as honest broker in future global negotiations on internet governance." `http://europa.eu/rapid/press-release_IP-14-142_en.htm`. 2014.

[6] "Amazon web services. whitepaper on eu data protection." `https://d0.awsstatic.com/whitepapers/compliance/AWS_EU_Data_Protection_Whitepaper.pdf`. 2015.

[7] F. Nawab, D. Agrawal, and A. El Abbadi, *Message futures: Fast commitment of transactions in multi-datacenter environments*, in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[8] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, *Minimizing commit latency of transactions in geo-replicated data stores*, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1279–1294, ACM, 2015.

[9] F. Nawab, D. Agrawal, A. El Abbadi, and S. Chawla, *COP: planning conflicts for faster parallel transactional machine learning*, in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pp. 132–143, 2017.

[10] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, *Chariots: A scalable shared log for data management in multi-datacenter cloud environments*, in *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pp. 13–24, 2015.

[11] J. Gray, *The transaction concept: Virtues and limitations (invited paper)*, in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pp. 144–154, VLDB Endowment, 1981.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] B. Kemme, R. Jimenez-Peris, and M. Patino-Martinez, *Database replication*, *Synthesis Lectures on Data Management* **2** (2010), no. 1 1–153.

[14] P. A. Bernstein and N. Goodman, *Concurrency control in distributed database systems*, *ACM Computing Surveys (CSUR)* **13** (1981), no. 2 185–221.

[15] H. T. Kung and J. T. Robinson, *On optimistic methods for concurrency control*, *ACM Trans. Database Syst.* **6** (June, 1981) 213–226.

[16] K. P. Eswaran *et. al.*, *The notions of consistency and predicate locks in a database system*, *Communications of the ACM* **19** (1976), no. 11 624–633.

[17] R. H. Thomas, *A majority consensus approach to concurrency control for multiple copy databases*, *ACM Transactions on Database Systems (TODS)* **4** (1979), no. 2 180–209.

[18] S. Ceri and S. Owicki, *On the use of optimistic methods for concurrency control in distributed databases*, in *Proceedings of the sixth Berkeley Conference on Distributed Data Management and Computer Network*, vol. 13452, p. 117, Lawrence Berkeley Laboratory, 1982.

[19] G. Schlageter, *Optimistic methods for concurrency control in distributed database systems*, in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pp. 125–130, VLDB Endowment, 1981.

[20] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta, *Distributed optimistic concurrency control with reduced rollback*, *Distributed Computing* **2** (Mar, 1987) 45–59.

[21] D. Agrawal and A. J. Bernstein, *A nonblocking quorum consensus protocol for replicated data*, *IEEE Transactions on Parallel and Distributed Systems* **2** (Apr, 1991) 171–179.

[22] M. Stonebraker, *Concurrency control and consistency of multiple copies of data in distributed ingres*, *IEEE Trans. Softw. Eng.* **5** (May, 1979) 188–194.

[23] P. A. Bernstein and N. Goodman, *Timestamp-based algorithms for concurrency control in distributed database systems*, in *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pp. 285–300, VLDB Endowment, 1980.

[24] O. T. Satyanarayanan and D. Agrawal, *Efficient execution of read-only transactions in replicated multiversion databases*, *IEEE Transactions on Knowledge and Data Engineering* **5** (Oct, 1993) 859–871.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, *Spanner: Google's globally-distributed database*, .

[26] L. Lamport, *The part-time parliament*, *ACM Trans. Comput. Syst.* **16** (May, 1998) 133–169.

[27] L. Lamport, *Paxos made simple*, *ACM SIGACT News* **32** (December, 2001) 18–25.

[28] L. Lamport, *Fast paxos*, *Distributed Computing* **19** (2006), no. 2 79–103.

[29] L. Lamport, *Generalized consensus and paxos*, tech. rep., Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[30] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, *Mdcc: Multi-data center consistency*, in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 113–126, ACM, 2013.

[31] H. Howard, D. Malkhi, and A. Spiegelman, *Flexible paxos: Quorum intersection revisited*, .

[32] I. Moraru, D. G. Andersen, and M. Kaminsky, *There is more consensus in egalitarian parliaments*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 358–372, ACM, 2013.

[33] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi, *Serializability, not serial: Concurrency control and availability in multi-datacenter datastores*, *Proc. VLDB Endow.* **5** (July, 2012) 1459–1470.

[34] Y. Mao, F. P. Junqueira, and K. Marzullo, *Mencius: Building efficient replicated state machines for wans*, in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 369–384, USENIX Association, 2008.

[35] W. Wei, H. T. Gao, F. Xu, and Q. Li, *Fast mencius: Mencius with low commit latency*, in *2013 Proceedings IEEE INFOCOM*, pp. 881–889, April, 2013.

[36] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, *S-paxos: Offloading the leader for high throughput state machine replication*, in *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 111–120, Oct, 2012.

[37] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing scalable, highly available storage for interactive services*, in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pp. 223–234, 2011.

[38] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, *Pnuts: Yahoo!'s hosted data serving platform*, *Proc. VLDB Endow.* **1** (Aug., 2008) 1277–1288.

[39] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's highly available key-value store*, in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[40] P. Bailis and A. Ghodsi, *Eventual consistency today: Limitations, extensions, and beyond*, *Queue* **11** (Mar., 2013) 20:20–20:32.

[41] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, *Commun. ACM* **21** (July, 1978) 558–565.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, *Don't settle for eventual: Scalable causal consistency for wide-area storage with cops*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 401–416, ACM, 2011.

[43] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, *Bolt-on causal consistency*, in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 761–772, ACM, 2013.

[44] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, *Stronger semantics for low-latency geo-replicated storage*, in *Proceedings of the 10th USENIX*

Conference on Networked Systems Design and Implementation, nsdi'13, (Berkeley, CA, USA), pp. 313–328, USENIX Association, 2013.

[45] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, *Orbe: Scalable causal consistency using dependency matrices and physical clocks*, in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 11:1–11:14, ACM, 2013.

[46] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, *A critique of ansi sql isolation levels*, .

[47] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, *Transactional storage for geo-replicated systems*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 385–400, ACM, 2011.

[48] K. Daudjee and K. Salem, *Lazy database replication with snapshot isolation*, in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pp. 715–726, VLDB Endowment, 2006.

[49] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, *Middleware based data replication providing snapshot isolation*, in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, (New York, NY, USA), pp. 419–430, ACM, 2005.

[50] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, *Enhancing edge computing with database replication*, in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, (Washington, DC, USA), pp. 45–54, IEEE Computer Society, 2007.

[51] J. Du, S. Elnikety, and W. Zwaenepoel, *Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks*, in *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, (Washington, DC, USA), pp. 173–184, IEEE Computer Society, 2013.

[52] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, *ACM Trans. Program. Lang. Syst.* **12** (July, 1990) 463–492.

[53] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, *Scalable consistency in scatter*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 15–28, ACM, 2011.

[54] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, *Low-latency multi-datacenter databases using replicated commit*, *Proc. VLDB Endow.* **6** (July, 2013) 661–672.

[55] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete, *Planet: Making progress with commit processing in unpredictable environments*, in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 3–14, ACM, 2014.

[56] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, *Coordination avoidance in database systems*, *Proc. VLDB Endow.* **8** (Nov., 2014) 185–196.

[57] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke, *The homeostasis protocol: Avoiding transaction coordination through program analysis*, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 1311–1326, ACM, 2015.

[58] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, *Transaction chains: Achieving serializability with low latency in geo-distributed storage systems*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 276–291, ACM, 2013.

[59] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi, *Db-risk: The game of global database placement*, in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), pp. 2185–2188, ACM, 2016.

[60] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, *Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 292–308, ACM, 2013.

[61] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, *Volley: Automated data placement for geo-distributed cloud services*, in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2010.

[62] P. T. Endo, A. V. de Almeida Palhares, N. N. Pereira, G. E. Goncalves, D. Sadok, J. Kelner, B. Melander, and J. E. Mangs, *Resource allocation for distributed cloud: concepts and research challenges*, *IEEE Network* **25** (July, 2011) 42–46.

[63] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, *Low latency geo-distributed data analytics*, in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 421–434, ACM, 2015.

[64] P. N. Shankaranarayanan, A. Sivakumar, S. Rao, and M. Tawarmalani, *Performance sensitive replication in geo-distributed cloud datastores*, in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, (Washington, DC, USA), pp. 240–251, IEEE Computer Society, 2014.

[65] A. Sharov, A. Shraer, A. Merchant, and M. Stokely, *Take me to your leader!: Online optimization of distributed storage configurations*, *Proc. VLDB Endow.* **8** (Aug., 2015) 1490–1501.

[66] Z. Wu, C. Yu, and H. V. Madhyastha, *Costlo: Cost-effective redundancy for lower latency variance on cloud storage services*, in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, (Berkeley, CA, USA), pp. 543–557, USENIX Association, 2015.

[67] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, *Consistency-based service level agreements for cloud storage*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 309–324, ACM, 2013.

[68] M. S. Ardekani and D. B. Terry, *A self-configurable geo-replicated cloud storage system*, in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 367–381, USENIX Association, 2014.

[69] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, *Millwheel: Fault-tolerant stream processing at internet scale*, *Proc. VLDB Endow.* **6** (Aug., 2013) 1033–1044.

[70] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, *Photon: Fault-tolerant and scalable joining of continuous data streams*, in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 577–588, ACM, 2013.

[71] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, and P. Tong, *F1: The fault-tolerant distributed rdbms supporting google's ad business*, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 777–778, ACM, 2012.

[72] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, *F1: A distributed sql database that scales*, *Proc. VLDB Endow.* **6** (Aug., 2013) 1068–1079.

[73] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal, *Mesa: Geo-replicated, near real-time, scalable data warehousing*, *Proc. VLDB Endow.* **7** (Aug., 2014) 1259–1270.

[74] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, *Storm@twitter*, in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[75] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, *Twitter heron: Stream processing at scale*, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 239–250, ACM, 2015.

[76] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, *Data warehousing and analytics infrastructure at facebook*, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 1013–1020, ACM, 2010.

[77] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang, *Data infrastructure at linkedin*, in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, (Washington, DC, USA), pp. 1370–1381, IEEE Computer Society, 2012.

[78] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, *Wanalytics: Analytics for a geo-distributed data-intensive world*, in *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[79] K. Y. Oktay, S. Mehrotra, V. Khadilkar, and M. Kantarcioglu, *Semrod: Secure and efficient mapreduce over hybrid clouds*, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 153–166, ACM, 2015.

[80] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, *Aggregation and degradation in jetstream: Streaming analytics in the wide area*, in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 275–288, USENIX Association, 2014.

[81] B. Heintz, A. Chandra, and R. K. Sitaraman, *Optimizing grouped aggregation in geo-distributed streaming analytics*, in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, (New York, NY, USA), pp. 133–144, ACM, 2015.

[82] A. Jonathan, A. Chandra, and J. B. Weissman, *Awan: Locality-aware resource manager for geo-distributed data-intensive applications*, in *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pp. 32–41, 2016.

[83] B. Heintz, A. Chandra, and R. K. Sitaraman, *Towards optimizing wide-area streaming analytics*, in *2015 IEEE International Conference on Cloud Engineering*, pp. 452–457, March, 2015.

[84] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola, *Towards geo-distributed machine learning*, vol. abs/1603.09035, 2016.

[85] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, *Pixida: Optimizing data parallel jobs in wide-area data analytics*, *Proc. VLDB Endow.* **9** (Oct., 2015) 72–83.

[86] C.-C. Hung, L. Golubchik, and M. Yu, *Scheduling jobs across geo-distributed datacenters*, in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, (New York, NY, USA), pp. 111–124, ACM, 2015.

[87] G. T. Wuu and A. J. Bernstein, *Efficient solutions to the replicated log and dictionary problems*, in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, (New York, NY, USA), pp. 233–242, ACM, 1984.

[88] "HBase." `http://hbase.apache.org`, 2011. [Online; acc. 18-Jul-2011].

[89] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking cloud serving systems with ycsb*, in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

[90] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, *Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration*, *Proc. VLDB Endow.* **4** (May, 2011) 494–505.

[91] D. L. Mills, *Internet time synchronization: the network time protocol*, *IEEE Transactions on Communications* **39** (Oct, 1991) 1482–1493.

[92] P. A. Bernstein, C. W. Reid, and S. Das, *Hyder - A transactional record manager for shared flash*, in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pp. 9–20, 2011.

[93] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, *Logbase: A scalable log-structured database system in the cloud*, *Proc. VLDB Endow.* **5** (June, 2012) 1004–1015.

[94] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, *Calvin: Fast distributed transactions for partitioned database systems*, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 1–12, ACM, 2012.

[95] D. P. Reed, *Naming and synchronization in a decentralized computer system*, tech. rep., Cambridge, MA, USA, 1978.

[96] D. Agrawal and V. Krishnaswamy, *Using multiversion data for non-interfering execution of write-only transactions*, in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, (New York, NY, USA), pp. 98–107, ACM, 1991.

[97] D. Agrawal and S. Sengupta, *Modular synchronization in multiversion databases: Version control and concurrency control*, in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, (New York, NY, USA), pp. 408–417, ACM, 1989.

[98] X. Pan, D. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan, *Parallel correlation clustering on big graphs*, in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, (Cambridge, MA, USA), pp. 82–90, MIT Press, 2015.

[99] X. Pan, J. Gonzalez, S. Jegelka, T. Broderick, and M. I. Jordan, *Optimistic concurrency control for distributed unsupervised learning*, in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, (USA), pp. 1403–1411, Curran Associates Inc., 2013.

[100] F. Niu, B. Recht, C. Re, and S. J. Wright, *Hogwild!: A lock-free approach to parallelizing stochastic gradient descent*, in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, (USA), pp. 693–701, Curran Associates Inc., 2011.

[101] X. Pan, S. Jegelka, J. Gonzalez, J. Bradley, and M. I. Jordan, *Parallel double greedy submodular maximization*, in *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, (Cambridge, MA, USA), pp. 118–126, MIT Press, 2014.

[102] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, *Malt: Distributed data-parallelism for existing ml applications*, in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), pp. 3:1–3:16, ACM, 2015.

[103] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, *Large scale distributed deep networks*, in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, (USA), pp. 1223–1231, Curran Associates Inc., 2012.

[104] X. Pan *et. al.*, *Scaling up correlation clustering through parallelism and concurrency control*, *In DISCML workshop at International Conference on Neural Information Processing Systems* (2014).

[105] A. Talwalkar, T. Kraska, R. Griffith, J. Duchi, J. Gonzalez, D. Britz, X. Pan, V. Smith, E. Sparks, A. Wibisono, *et. al.*, *Mlbase: A distributed machine learning wrapper*, *Big Learning Workshop at NIPS* (2012).

[106] C. D. Sa, C. Zhang, K. Olukotun, and C. Ré, *Taming the wild: A unified analysis of hog wild! -style algorithms*, in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, (Cambridge, MA, USA), pp. 2674–2682, MIT Press, 2015.

[107] J. Gray and A. Reuter, *Transaction processing: concepts and techniques*. Elsevier, 1992.

[108] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, *Mlbase: A distributed machine-learning system*, in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[109] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, *The weka data mining software: An update*, *SIGKDD Explor. Newsl.* **11** (Nov., 2009) 10–18.

[110] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, *Graphlab: A new framework for parallel machine learning*, *arXiv preprint arXiv:1408.2041* (2014).

[111] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *Powergraph: Distributed graph-parallel computation on natural graphs*, in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2012.

[112] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, *Speedy transactions in multicore in-memory databases*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 18–32, ACM, 2013.

[113] J. Stamper, A. Niculescu-Mizil, S. Ritter, G. Gordon, and K. Koedinger, *Challenge data set from kdd cup 2010 educational data mining challenge*, 2010. [https://pslcdatashop.web.cmu.edu/KDDCup/].

[114] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, and C. Ré, *Cyclades: Conflict-free asynchronous machine learning*, in *Advances in Neural Information Processing Systems 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 2568–2576. Curran Associates, Inc., 2016.

[115] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar, *An asynchronous parallel stochastic coordinate descent algorithm*, *J. Mach. Learn. Res.* **16** (Jan., 2015) 285–322.

[116] H. Avron, A. Druinsky, and A. Gupta, *Revisiting asynchronous linear solvers: Provable convergence rate through randomization*, *J. ACM* **62** (Dec., 2015) 51:1–51:27.

[117] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, *More effective distributed ml via a stale synchronous parallel parameter server*, in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, (USA), pp. 1223–1231, Curran Associates Inc., 2013.

[118] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, *Scaling distributed machine learning with the parameter server*, in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 583–598, USENIX Association, 2014.

[119] C. Curino, E. Jones, Y. Zhang, and S. Madden, *Schism: A workload-driven approach to database replication and partitioning*, *Proc. VLDB Endow.* **3** (Sept., 2010) 48–57.

[120] "P. Helland. Immutability changes everything!." `http://vimeo.com/52831373`. Talk at RICON, 2012.

[121] "N. Marz. How to beat the CAP theorem." `http://bit.ly/marz-cap-theorem`.

[122] "J. Bonér. The Road to Akka Cluster and Beyond." `https://www.youtube.com/watch?v=2wSYcyWCtx4`.

[123] N. Conway, P. Alvaro, E. Andrews, and J. M. Hellerstein, *Edelweiss: Automatic storage reclamation for distributed programming*, *Proc. VLDB Endow.* **7** (Feb., 2014) 481–492.

[124] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, *Corfu: A shared log design for flash clusters*, in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2012.

[125] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, *Tango: Distributed data structures over a shared log*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 325–340, ACM, 2013.

[126] E. A. Brewer, *Towards robust distributed systems (abstract)*, in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, (New York, NY, USA), pp. 7–, ACM, 2000.

[127] S. Gilbert and N. Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, *SIGACT News* **33** (June, 2002) 51–59.

[128] A. Lakshman and P. Malik, *Cassandra: A decentralized structured storage system*, *SIGOPS Oper. Syst. Rev.* **44** (Apr., 2010) 35–40.

[129] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, *Providing high availability using lazy replication*, *ACM Trans. Comput. Syst.* **10** (Nov., 1992) 360–391.

[130] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, *Practi replication*, in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2006.

[131] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, *Flexible update propagation for weakly consistent replication*, *SIGOPS Oper. Syst. Rev.* **31** (Oct., 1997) 288–301.

[132] P. Mahajan, L. Alvisi, and M. Dahlin, *Consistency, availability, and convergence*, *University of Texas at Austin Tech Report* **11** (2011).

[133] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, *Fast crash recovery in ramcloud*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 29–41, ACM, 2011.

[134] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan, *Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications*, in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 257–273, USENIX Association, 2014.

[135] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, *Ivy: A read/write peer-to-peer file system*, *SIGOPS Oper. Syst. Rev.* **36** (Dec., 2002) 31–44.

[136] J. H. Hartman and J. K. Ousterhout, *The zebra striped network file system*, *ACM Trans. Comput. Syst.* **13** (Aug., 1995) 274–310.

[137] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in bayou, a weakly connected replicated storage system*, in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 172–182, ACM, 1995.

[138] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, *The recovery manager of the system r database manager*, *ACM Comput. Surv.* **13** (June, 1981) 223–242.

[139] M. Stonebraker and L. A. Rowe, *The design of postgres*, in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, (New York, NY, USA), pp. 340–355, ACM, 1986.

[140] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, *Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging*, *ACM Trans. Database Syst.* **17** (Mar., 1992) 94–162.

[141] M. Rosenblum and J. K. Ousterhout, *The design and implementation of a log-structured file system*, *ACM Trans. Comput. Syst.* **10** (Feb., 1992) 26–52.

[142] M. Vrable, S. Savage, and G. M. Voelker, *Bluesky: A cloud-backed file system for the enterprise*, in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 2012.

[143] L. A. Barroso and U. Hölzle, *The datacenter as a computer: An introduction to the design of warehouse-scale machines*, *Synthesis lectures on computer architecture* **4** (2009), no. 1 1–108.

[144] S. Wang, D. Maier, and B. C. Ooi, *Lightweight indexing of observational data in log-structured storage*, *Proc. VLDB Endow.* **7** (Mar., 2014) 529–540.

[145] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, *Fog computing and its role in the internet of things*, in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

[146] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, *The case for vm-based cloudlets in mobile computing*, *IEEE pervasive Computing* **8** (2009), no. 4 14–23.

[147] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat, *Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing*, in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 1–14, ACM, 2015.

[148] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, *B4: Experience with a globally-deployed software defined wan*, in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 3–14, ACM, 2013.